

ALAGAPPA UNIVERSITY



[Accredited with 'A+' Grade by NAAC (CGPA:3.64) in the Third Cycle
and Graded as Category–I University by MHRD-UGC]

(A State University Established by the Government of Tamil Nadu)

KARAIKUDI – 630 003

Directorate of Distance Education

M.Sc. (Information Technology)

IV - Semester

313 42

SOFTWARE ENGINEERING

Author

Rohit Khurana, *CEO, ITL Education solutions Ltd.*

Units (1-14)



Vikas® is the registered trademark of Vikas® Publishing House Pvt. Ltd.

VIKAS® PUBLISHING HOUSE PVT. LTD.
E-28, Sector-8, Noida - 201301 (UP)
Phone: 0120-4078900 • Fax: 0120-4078999
Regd. Office: A-27, 2nd Floor, Mohan Co-operative Industrial Estate, New Delhi 1100 44
• Website: www.vikaspublishing.com • Email: helpline@vikaspublishing.com

Work Order No. AU/DDE/DE12-16/Preparation and Printing of Course Materials/2020 Dated 28.02.2020 Copies - 500

SYLLABI-BOOK MAPPING TABLE

Software Engineering

Syllabi	Mapping in Book
BLOCK I : INTRODUCTION **Unit 1 Software:** Role of Software, Software Myths. Generic View of Process: A Layered Technology, A Process Framework, The Capability Maturity Model Integration (CMMI). **Unit 2 Process Patterns:** Process Assessment, Personal and Team Process Models. **Unit 3 Process Model:** The Waterfall Model, Incremental Process Models, Evolutionary Process Models, The Unified Process.	**Unit 1:** Software **(Pages 1-14);** **Unit 2:** Process Patterns **(Pages 15-24);** **Unit 3:** Process Model **(Pages 25-36)**
BLOCK II : REQUIREMENT ENGINEERING **Unit 4 Requirement Engineering–Design and Construction:** Design and Construction, Requirement Engineering Tasks, Requirements Engineering Process, Validating Requirements. **Unit 5 Building the Analysis Model:** Requirement Analysis, Data Modeling Concepts, Object-Oriented Analysis. **Unit 6 Modeling:** Scenario-Based Modeling, Flow-Oriented Modeling, Class-Based Modeling, Creating a Behavioral Model.	**Unit 4:** Requirement Engineering–Design and Construction **(Pages 37-53);** **Unit 5:** Building the Analysis Model **(Pages 54-68);** **Unit 6:** Modeling **(Pages 69-81)**
BLOCK III : DESIGN **Unit 7 Design Engineering:** Design Process and Quality, Design Concepts, The Design Model. **Unit 8 Architectural Design:** Software Architecture, Data Design, Architectural Styles and Patterns, Architectural Design. **Unit 9 User Interface Design:** The Golden rules, User Interface Analysis and Design, Interface Analysis, Interface Design Steps, Design Evaluation.	**Unit 7:** Design Engineering **(Pages 82-94);** **Unit 8:** Architectural Design **(Pages 95-109);** **Unit 9:** User Interface Design **(Pages 110-119)**
BLOCK IV : TESTING **Unit 10 Testing Strategies:** Approach to Software Testing, Unit Testing, Integration Testing, Test Strategies for Object-Oriented Software, Validation Testing, System Testing, the Art of Debugging, Black-Box and White-Box Testing. **Unit 11 Product Metrics:** Software Quality, Product Metrics, Metrics for Analysis Model, Design Model, Source Code and Metrics for Testing, Metrics for Maintenance. Metrics for Process and Projects Domains: Software Measurement, Metrics for Software Quality and Software Process.	**Unit 10:** Testing Strategies **(Pages 120-176);** **Unit 11:** Product Metrics **(Pages 177-202)**
BLOCK V : RISK AND QUALITY MANAGEMENT **Unit 12 Risk Strategies:** Reactive vs. Proactive Risk Strategies, Software Risks, Risk Identification **Unit 13 Risk Protection and Refinement:** Risk Projection, Risk Refinement, Risk Mitigation, Monitoring and Management, RMMM Plan. **Unit 14 Quality Management:** Quality Concepts, Software Quality Assurance, Software Reviews, Formal Technical Reviews, Statistical Software Quality Assurance, Software Reliability, The ISO 9000 Quality Standards.	**Unit 12:** Risk Strategies **(Pages 203-208);** **Unit 13:** Risk Projection and Refinement **(Pages 209-214);** **Unit 14:** Quality Management **(Pages 215-240)**

CONTENTS

INTRODUCTION

NOTES

The notion of software engineering was first proposed in 1968. Since then, software engineering has evolved as a full-fledged engineering discipline that is accepted as a field involving in-depth study and research. Software engineering methods and tools have been successfully implemented in various applications spread across different walks of life. Software engineering has been defined as a systematic approach to develop software within a specified time and budget. It provides methods to handle complexities in a software system and enables the development of reliable software systems that maximize productivity.

The Institute of Electrical and Electronic Engineers (IEEE) defines software as 'A collection of computer programs, procedures, rules and associated documentation and data'. Software is responsible for managing, controlling and integrating the hardware components of a computer system in order to accomplish a specific task. It tells the computer what to do and how to do it.

This book discusses the phases of software development, the design process, the coding methodology and problems that might occur during the development procedure of software. Software engineering offers ways and means to deal with intricacies in a software system and facilitates the development of dependable software systems, which maximize productivity. Software development comprises two phases: Requirement Analysis and Planning. The requirement analysis phase measures the requirements of the end-user, while planning chalks out the strategy to achieve those requirements through the software to be produced. Cost estimation establishes an estimate of the budget required to develop a software or project and also helps in effective use of resources and time. This is done at the beginning of the project and also between various stages of development to capture any fluctuation of expenditure.

Once the requirements are specified, the design process begins. In this process, the software engineer collates the customer's business requirements and technical considerations to model the product to be built. After successful completion of the design phase, the specifications thus created are translated into source code. This is the implementation phase wherein the language that meets user's requirements in the best way is selected for coding.

The software is tested after the implementation phase culminates. Testing brings forward any kind of errors and bugs that exist, evaluates the capability of the system and ensures whether the system is built to meet the target users' requirements. The more stringent the testing, the better will be the quality of the product. As the product is used in the market, the users' requirements keep on changing. This calls for additions of new features and functionalities. Software maintenance modifies the system according to the user's needs and also eliminates errors, as and when they arise.

This book, *Software Engineering*, follows the SIM format or the Self-Instructional Mode wherein each unit begins with an 'Introduction' to the topic followed by an outline of the 'Objectives'. The detailed content is then presented in a simple and organized manner, interspersed with 'Check Your Progress' questions to test the understanding of the students. A 'Summary' along with a list of 'Key Words' and a set of 'Self Assessment Questions and Exercises' is also provided at the end of each unit for effective recapitulation.

BLOCK - I
INTRODUCTION

NOTES

UNIT 1 SOFTWARE

Structure

1.0 INTRODUCTION

In this unit, you will learn about the software and software development project. In earlier times, software was simple in nature and hence, software development was a simple activity. However, as technology improved, software became more complex and software projects grew larger. Software development now necessitated the presence of a team, which could prepare detailed plans and designs, carry out testing, develop intuitive user interfaces, and integrate all these activities into a system. This new approach led to the emergence of a discipline known as software engineering.

Software engineering provides methods to handle complexities in a software system and enables the development of reliable software systems, which maximize productivity. In addition to the technical aspects of the software development, it also covers management activities which include guiding the team, budgeting, preparing schedules, etc. The notion of software engineering was first proposed in 1968. Since then, software engineering has evolved as a full-fledged engineering discipline, which is accepted as a field involving in-depth study and research. Software engineering methods and tools have been successfully implemented in various applications spread across different walks of life.

1.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain software characteristics and classification of software

NOTES

- Understand software myths such as those related to management, users, and developers
- Discuss software crisis, the situation where catastrophic failures have occurred
- Discuss the software process components and process framework

1.2 ROLE OF SOFTWARE

Software is defined as a collection of programs, documentation and operating procedures. The **Institute of Electrical and Electronic Engineers** (**IEEE**) defines software as a '*collection of computer programs, procedures, rules and associated documentation and data*.' It possesses no mass, no volume, and no colour, which makes it a non-degradable entity over a long period. Software does not wear out or get tired.

Software controls, integrates, and manages the hardware components of a computer system. It also instructs the computer what needs to be done to perform a specific task and how it is to be done. For example, software instructs the hardware how to print a document, take input from the user, and display the output.

Computer works only in response to instructions provided externally. Usually, the instructions to perform some intended tasks are organized into a program using a programming language like C, C++, Java, etc., and submitted to computer. Computer interprets and executes these instructions and provides response to the user accordingly. A set of programs intended to provide users with a set of interrelated functionalities is known as a **software package**. For example, an accounting software package such as Tally provides users the functionality to perform accounting-related activities.

Software Characteristics

Different individuals judge software on different basis. This is because they are involved with the software in different ways. For example, users want the software to perform according to their requirements. Similarly, developers involved in designing, coding, and maintenance of the software evaluate the software by looking at its internal characteristics, before delivering it to the user. Software characteristics are classified into six major components, which are shown in Figure 1.1.



Fig. 1.1 *Software Characteristics*

NOTES

- **Functionality:** Refers to the degree of performance of the software against its intended purpose.
- **Reliability:** Refers to the ability of the software to provide desired functionality under the given conditions.
- **Usability:** Refers to the extent to which the software can be used with ease.
- **Efficiency:** Refers to the ability of the software to use system resources in the most effective and efficient manner.
- **Maintainability:** Refers to the ease with which the modifications can be made in a software system to extend its functionality, improve its performance, or correct errors.
- **Portability:** Refers to the ease with which software developers can transfer software from one platform to another, without (or with minimum) changes. In simple terms, it refers to the ability of software to function properly on different hardware and software platforms without making any changes in it.

In addition to the above mentioned characteristics, robustness and integrity are also important. **Robustness** refers to the degree to which the software can keep on functioning in spite of being provided with invalid data while **integrity** refers to the degree to which unauthorized access to the software or data can be prevented.

Classification of Software

Software can be applied in countless fields such as business, education, social sector, and other fields. It is designed to suit some specific goals such as data processing, information sharing, communication, and so on. It is classified according to the range of potential of applications. These classifications are listed below.

- **System software:** This class of software manages and controls the internal operations of a computer system. It is a group of programs, which is responsible for using computer resources efficiently and effectively. For example, an operating system is a system software, which controls the hardware, manages memory and multitasking functions, and acts as an interface between application programs and the computer.
- **Real-time software:** This class of software observes, analyzes, and controls real world events as they occur. Generally, a real-time system guarantees a response to an external event within a specified period of time. An example of real-time software is the software used for weather forecasting that collects and processes parameters like temperature and humidity from the external environment to forecast the weather. Most of the defence organizations all over the world use real-time software to control their military hardware.

NOTES

- **Business software:** This class of software is widely used in areas where management and control of financial activities is of utmost importance. The fundamental component of a business system comprises payroll, inventory, and accounting software that permit the user to access relevant data from the database. These activities are usually performed with the help of specialized business software that facilitates efficient framework in business operations and in management decisions.
- **Engineering and scientific software:** This class of software has emerged as a powerful tool in the research and development of next generation technology. Applications such as the study of celestial bodies, under-surface activities, and programming of an orbital path for space shuttles are heavily dependent on engineering and scientific software. This software is designed to perform precise calculations on complex numerical data that are obtained during real-time environment.
- **Artificial intelligence (AI) software:** This class of software is used where the problem-solving technique is non-algorithmic in nature. The solutions of such problems are generally non-agreeable to computation or straightforward analysis. Instead, these problems require specific problem-solving strategies that include expert system, pattern recognition, and game-playing techniques. In addition, they involve different kinds of search techniques which include the use of heuristics. The role of artificial intelligence software is to add certain degrees of intelligence to the mechanical hardware in order to get the desired work done in an agile manner.
- **Web-based software:** This class of software acts as an interface between the user and the Internet. Data on the Internet is in the form of text, audio, or video format, linked with hyperlinks. Web browser is a software that retrieves web pages from the Internet. The software incorporates executable instructions written in special scripting languages such as CGI or ASP. Apart from providing navigation on the Web, this software also supports additional features that are useful while surfing the Internet.
- **Personal computer (PC) software:** This class of software is used for both official and personal use. The personal computer software market has grown over in the last two decades from normal text editor to word processor and from simple paintbrush to advanced image-editing software. This software is used predominantly in almost every field, whether it is database management system, financial accounting package, or multimedia-based software. It has emerged as a versatile tool for routine applications.

1.2.1 Software Myths

The development of software requires dedication and understanding on the developers' part. Many software problems arise due to myths that are formed during the initial stages of software development. Unlike ancient folklore that often provides valuable lessons, software myths propagate false beliefs and confusion in the minds of management, users and developers.

Management Myths

NOTES

Managers, who own software development responsibility, are often under strain and pressure to maintain a software budget, time constraints, improved quality, and many other considerations. Common management myths are listed in Table 1.1.

Table 1.1 *Management Myths*

Myths	Realities
• The members of an organization can acquire all the information they require from a manual, which contains standards, procedures, and principles.	• Standards are often incomplete, inadaptable, and outdated. • Developers are often unaware of all the established standards. • Developers rarely follow all the known standards because not all the standards tend to decrease the delivery time of software while maintaining its quality.
• If the project is behind schedule, increasing the number of programmers can reduce the time gap.	• Adding more manpower to the project, which is already behind schedule, further delays the project. • New workers take longer to learn about the project as compared to those already working on the project.
• If the project is outsourced to a third party, the management can relax and let the other firm develop software for them.	• Outsourcing software to a third party does not help the organization, which is incompetent in managing and controlling the software project internally. The organization invariably suffers when it outsources the software project.

User Myths

In most cases, users tend to believe myths about the software because software managers and developers do not try to correct the false beliefs. These myths lead to false expectations and ultimately develop dissatisfaction among the users. Common user myths are listed in Table 1.2.

Table 1.2 *User Myths*

Myths	Realities
• Brief requirement stated in the initial process is enough to start development; detailed requirements can be added at the later stages.	• Starting development with incomplete and ambiguous requirements often lead to software failure. Instead, a complete and formal description of requirements is essential before starting development. • Adding requirements at a later stage often requires repeating the entire development process.
• Software is flexible; hence software requirement changes can be added during any phase of the development process.	• Incorporating change requests earlier in the development process costs lesser than those that occurs at later stages. This is because incorporating changes later may require redesigning and extra resources.

NOTES

Developer Myths

In the early days of software development, programming was viewed as an art, but now software development has gradually become an engineering discipline. However, developers still believe in some myths. Some of the common developer myths are listed in Table 1.3.

Table 1.3 *Developer Myths*

Myths	Realities
• Software development is considered complete when the code is delivered.	• 50% to 70% of all the effort are expended after the software is delivered to the user.
• The success of a software project depends on the quality of the product produced.	• The quality of programs is not the only factor that makes the project successful instead the documentation and software configuration also play a crucial role.
• Software engineering requires unnecessary documentation, which slows down the project.	• Software engineering is about creating quality at every level of the software project. Proper documentation enhances quality which results in reducing the amount of rework.
• The only product that is delivered after the completion of a project is the working program(s).	• The deliverables of a successful project includes not only the working program but also the documentation to guide the users for using the software.
• Software quality can be assessed only after the program is executed.	• The quality of software can be measured during any phase of development process by applying some quality assurance mechanism. One such mechanism is formal technical review that can be effectively used during each phase of development to uncover certain errors.

Software Crisis

In the late 1960s, it became clear that the development of software is different from manufacturing other products. This is because employing more manpower (programmers) later in the software development does not always help speed up the development process. Instead, sometimes it may have negative impacts like delay in achieving the scheduled targets, degradation of software quality, etc. Though software has been an important element of many systems since a long time, developing software within a certain schedule and maintaining its quality is still difficult.

History has seen that delivering software after the scheduled date or with errors has caused large scale financial losses as well as inconvenience to many. Disasters such as the Y2K problem affected economic, political, and administrative systems of various countries around the world. This situation, where catastrophic failures have occurred, is known as **software crisis**. The major causes of software crisis are the problems associated with poor quality software such as malfunctioning

of software systems, inefficient development of software, and the most important, dissatisfaction amongst the users of the software.

NOTES

The software market today has a turnover of more than millions of rupees. Out of this, approximately thirty Percent of software is used for personal computers and the remaining software is developed for specific users or organizations. Application areas such as the banking sector are completely dependant on software application. Software failures in these technology-oriented areas have led to considerable loss in terms of time, money, and even human lives. History has been witness to many such failures, some of which are listed below.

- The Northeast blackout in 2003 has been one of the major power system failures in the history of North America. This blackout involved failure of 100 power plants due to which almost 50 million customers faced power loss that resulted in financial loss of approximately $6 billion. Later, it was determined that the major reason behind the failure was a software bug in the power monitoring and management system.
- Year 2000 (Y2K) problem refers to the widespread snags in processing dates after the year 2000. The roots of Y2K problem can be traced back to 1960-80 when developers shortened the 4-digit date format like 1972 to a 2-digit format like 72 because of limited memory. At that time they did not realize that year 2000 will be shortened to 00 which is less than 72. In the 1990s, experts began to realize this major shortcoming in the computer application and then millions were spent to handle this problem.
- In 1996, Arian-5 space rocket, developed at the cost of $7000 million over a period of 10 years was destroyed within less than a minute after its launch. The crash occurred because there was a software bug in the rocket guidance system.
- In 1996, one of the largest banks of US credited accounts of nearly 800 customers with approximately $924 lacs. Later, it was detected that the problem occurred due to a programming bug in the banking software.
- During the Gulf War in 1991, the United States of America used Patriot missiles as a defense against Iraqi Scud missiles. However, the Patriot failed to hit the Scud many times. As a result, 28 US soldiers were killed in Dhahran, Saudi Arabia. An inquiry into the incident concluded that a small bug had resulted in the miscalculation of missile path.

1.3 GENERIC VIEW OF PROCESS

Software engineering comprises interrelated and recurring entities, which are essential for software development. Software is developed efficiently and effectively with the help of well defined activities or processes. In general, a process is defined as a series of steps involving activities and resources, which produces the desired

NOTES

output. Software process is defined as a collection of procedures to develop the software product according to certain goals or standards. Generally, the following points are noted about software process.

- It uses resources subject to given constraints and produce intermediate and final products.
- It is composed of sub processes that are organized in such a manner that each sub process has its own process model.
- It is carried out with an entry and exit criteria that helps in monitoring the beginning and completion of the activity.
- It includes guidelines, which explain the objectives of each activity.
- It is vital because it imposes uniformity on the set of activities.
- It is regarded as more than just a procedure, tools or techniques, which are collectively used in a structured manner to produce a product.
- It includes various technical and managerial issues, which are required to develop the software.

'The characteristics of software processes are listed in Table 1.4.

Table 1.4 *Software Process Characteristics*

Characteristic	***Description***
Understandability	The extent to which the process is explicitly defined and the ease with which the process definition is understood.
Visibility	Whether the process activities culminate in clear results so that the progress of the process is visible externally.
Supportability	The extent to which CASE tools can support the process activities.
Acceptability	The extent to which the defined process is acceptable and usable by the engineers, responsible for producing the software product.
Reliability	The manner in which the process is designed such that errors in the process are avoided or trapped before they result in errors in the product.
Robustness	Whether the process can continue in spite of unexpected problems.
Maintainability	Whether the process can evolve to reflect the changing organizational requirements or identify process improvements.
Rapidity	The speed with which the complete software can be delivered with given specifications.

A **project** is defined as a specification essential for developing or maintaining a specific product. Software project is developed when software processes are

NOTES

executed for certain specific requirements of the user. Thus, by using software process, the software project can be easily developed. The activities in software project comprise various tasks for managing resources and developing products. Figure 1.2 shows that software project involves people (developers, project manager, end-users, and so on) also referred to as participants who use software processes to create a product according to the user's requirements. The participants play a major role in the development of the project and select the appropriate process for the project. In addition, a project is efficient if it is developed within the time constraint. The outcome or the result of the software project is known as a **product**. Thus, a software project uses software processes to create a product.

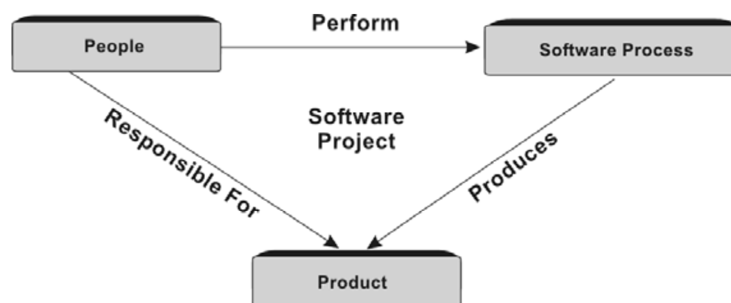


Fig. 1.2 *Software Project*

Software process can be used in the development of many software projects, each of which can produce one or more software products. The interrelationship among these three entities (process, project, and product) is shown in Figure 1.3. Software project begins with requirements and ends with the accomplishment of the requirements. Thus, the software process should be performed to develop the final software by accomplishing user's requirements.

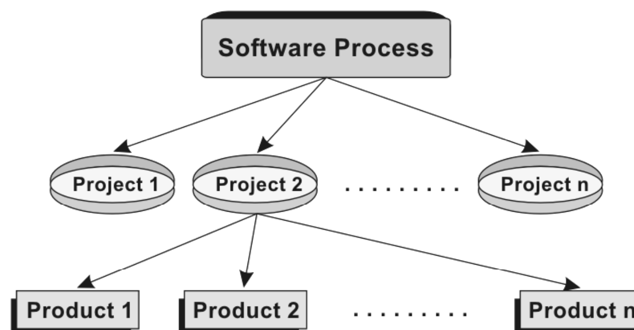


Fig. 1.3 *Process, Projects, and Products*

Software Process Components

The objective of the software process is to develop a product, which accomplishes user's requirements. For this, software processes require components, which are shown in Figure 1.4. The major components of the software process include a process management process and a product engineering process. The **Process Management Process** (**PMP**) aims at improving software processes so that a

NOTES

cost effective and high-quality product is developed. For this, the existing processes of the completed projects are examined. The process of comprehending the existing process, analyzing its properties, determining how to improve it, and then effecting the improvement is carried out by PMP. A group known as the **Software Engineering Process Group** (**SEPG**) performs the activities of the process management.

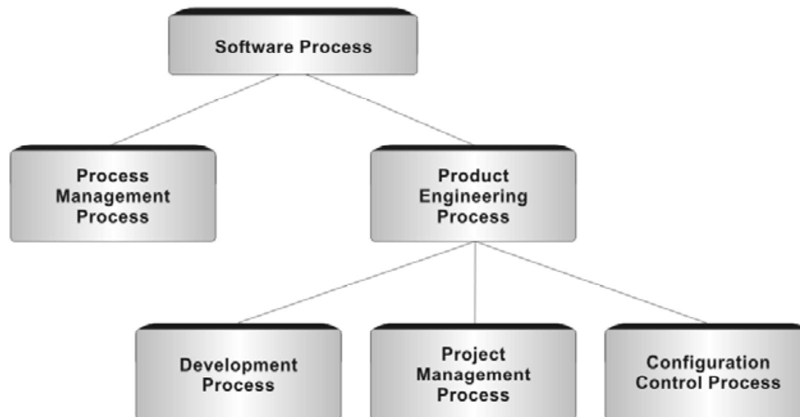


Fig. 1.4 *Components of Software Process*

Based on above mentioned analysis, the product engineering processes are improved, thereby improving the software process. The aim of the product engineering process is to develop the product according to the user's requirements. The product engineering process comprises three major components, which are listed below.

- **Development process:** It is the process which is used during the development of the software. It specifies the development and quality assurance activities that are to be performed. Programmers, designers, testing personnel, etc., perform these processes.
- **Project management process:** It is concerned with the set of activities or tasks, which are used to successfully accomplish a set of goals. It provides the means to plan, organize, and control the allocated resources to accomplish project costs, time, and performance objectives. For this, various processes, techniques, and tools are used to achieve the objectives of the project. Project management team performs the activities of this process.
- **Configuration control process:** It manages changes that occur as a result of modifying the requirements. In addition, it maintains integrity of the products with the changed requirements. The activities in configuration control processes are performed by a group called the **Configuration Control Board** (**CCB**).

Note that the project management process and configuration control process depend on the development process. The management process aims to control the development process, depending on the activities in the development process.

NOTES

1.3.1 Process Framework

Process framework (see Figure 1.5) determines the processes which are essential for completing a complex software project. This framework identifies certain activities, known as **framework activities**, which are applicable to all software projects regardless of their type and complexity. Some of the framework activities are listed below.

- **Communication:** It involves communication with the user so that the requirements are easily understood.
- **Planning:** It establishes a plan for accomplishing the project. It describes the schedule for the project, the technical tasks involved, expected risks, and the required resources.
- **Modeling:** It encompasses creation of models, which allow the developer and the user to understand software requirements and the designs to achieve those requirements.
- **Construction:** It combines generation of code with testing to uncover errors in the code.
- **Deployment:** It implies that the final product (software) is delivered to the user. The user evaluates the delivered product and provides feedback based on the evaluation.

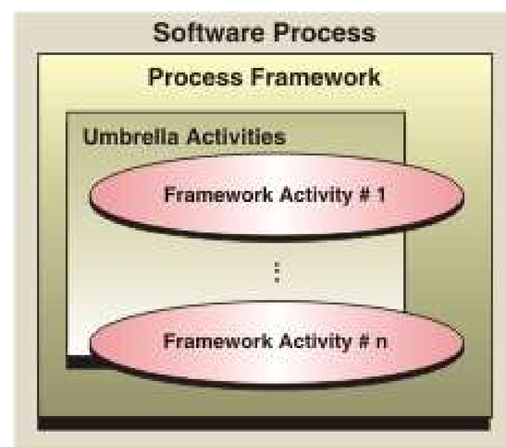


Fig. 1.5 *Process Framework*

In addition to framework activities, process framework also comprises a set of activities known as **umbrella activities**, which are used throughout the software process. These activities are listed below.

- **Software project tracking and control:** It monitors the actual process so that management can take necessary steps if the software project deviates from the plans laid down. It involves tracking procedures and reviews to check whether the software project is according to user's requirements. A documented plan is used as a basis for tracking the software activities and revising the plans.

NOTES

- **Formal technical reviews:** It assesses the code, products and documents of software engineering practices to detect and remove errors.
- **Software quality assurance:** It assures that the software is according to the requirements. In addition, it is designed to evaluate the processes of developing and maintaining quality of the software.
- **Reusability management:** It determines the criteria for products' reuse and establishes mechanisms to achieve reusable components.
- **Software configuration management:** It manages the changes made in the software processes of the products throughout the life cycle of the software project. It controls changes made to the configuration and maintains the integrity in the software development process.
- **Risk management:** It identifies, analyzes, evaluates, and eliminates the possibility of unfavourable deviations from expected results, by following a systematic activity and then develops strategies to manage them.

1.3.2 Capability Maturity Model Integration (CMMI)

It is a process level improvement training and appraisal program. Administered by the CMMI Institute which is a subsidiary of ISACA. It was developed at Carnegie Mellon University (CMU). An appraisal examines processes to determine their strengths and weaknesses in an organization. The appraisal used in an organization evaluates the internal processes (series of procedures occurring in the organization) for establishing or updating process improvement. This model is explained in detail in unit 2.

Check Your Progress

1. Define software.
2. What is software package?
3. What is system software?

1.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Software is defined as a collection of programs, documentation and operating procedures.
2. A set of programs intended to provide users with a set of interrelated functionalities is known as a software package.
3. System software is a type of computer program that is designed to run a computer's hardware and application programs.

1.5 SUMMARY

- Software is defined as a collection of programs, documentation and operating procedures. The Institute of Electrical and Electronic Engineers (IEEE) defines software as a 'collection of computer programs, procedures, rules and associated documentation and data'.
- Software can be applied in countless fields such as business, education, social sector, and other fields. It is designed to suit some specific goals such as data processing, information sharing, communication, and so on.
- Artificial intelligence (AI) software is used where the problem-solving technique is non-algorithmic in nature. The solutions of such problems are generally non-agreeable to computation or straightforward analysis.
- Software engineering comprises interrelated and recurring entities, which are essential for software development. Software is developed efficiently and effectively with the help of well-defined activities or processes.
- A project is defined as a specification essential for developing or maintaining a specific product. Software project is developed when software processes are executed for certain specific requirements of the user.
- Software process can be used in the development of many software projects, each of which can produce one or more software products.
- Process framework determines the processes which are essential for completing a complex software project. This framework identifies certain activities, known as framework activities.

NOTES

1.6 KEY WORDS

- **System Software:** This class of software manages and controls the internal operations of a computer system.
- **Business Software:** This class of software is widely used in areas where management and control of financial activities is of utmost importance.
- **Real-time Software:** This class of software observes, analyzes, and controls real world events as they occur. Generally, a real-time system guarantees a response to an external event within a specified period of time.

1.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Discuss the characteristics of software.

NOTES

2. What do you understand by software crisis?
3. Discuss the significance of process components.

Long Answer Questions

1. Explain the different types of software.
2. What do you understand by software myths? Explain.
3. Write a detailed note on process framework.

1.8 FURTHER READINGS

Pressman, Roger S. 1997. *Software Engineering, a Practitioner's Approach*. New Delhi: Tata McGraw-Hill.

Jalote, Pankaj. 1991. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

Schach, Stephen R. 2005. *Object Oriented and Classical Software Engineering*. New Delhi: Tata McGraw-Hill.

Somerville, Ian. 2001. *Software Engineering*. New Delhi: Pearson Education.

Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandriolli . 1991. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hill of India.

Jawadekar, Waman S. 2004. *Software Engineering: Principles and Practice*. New Delhi: Tata McGraw-Hill.

UNIT 2 PROCESS PATTERNS

NOTES

Structure

2.0 INTRODUCTION

In this unit, you will learn about the process assessment. To accomplish a set of tasks, it is important to go through a sequence of predictable steps. This sequence of steps refers to a road map, which helps in developing a timely, high quality, and highly efficient product or system. Road map, commonly referred to as software process, comprises activities, constraints, and resources that are used to produce an intended system. Software process helps to maintain a level of consistency and quality in products or services that are produced by different people. The process needs to be assessed in order to ensure that it meets a set of basic process criteria, which is essential for implementing the principles of software engineering in an efficient manner. You will also learn about the personal and team process models.

2.1 OBJECTIVES

After going through this unit, you will be able to:

- Discuss the need for process assessment to ensure that it meets a set of basic process criteria
- Understand the different approaches used for assessing software process
- Explain personal and team process models

2.2 PROCESS ASSESSMENT

The existence of software process does not guarantee the timely delivery of the software and its ability to meet the user's expectations. The process needs to be assessed in order to ensure that it meets a set of basic process criteria, which is essential for implementing the principles of software engineering in an efficient manner. The process is assessed to evaluate methods, tools, and practices, which

NOTES

are used to develop and test the software. The aim of process assessment is to *identify* the areas for improvement and suggest a plan for making that improvement. The main focus areas of process assessment are listed below.

- Obtaining guidance for improving software development and test processes
- Obtaining an independent and unbiased review of the process
- Obtaining a baseline (defined as a set of software components and documents that have been formerly reviewed and accepted; that serves as the basis for further development) for improving quality and productivity of processes.

As shown in Figure 2.1, software process assessment examines whether the software processes are effective and efficient in accomplishing the goals. This is determined by the capability of selected software processes. The capability of a process determines whether a process with some variations is capable of meeting user's requirements. In addition, it measures the extent to which the software process meets the user's requirements. Process assessment is useful to the organization as it helps in improving the existing processes. In addition, it determines the strengths, weaknesses and the risks involved in the processes.



Fig. 2.1 *Software Process Assessment*

Figure 2.1 also shows that process assessment leads to process capability determination and process improvement. Process capability determination is an organized assessment, which analyzes the software processes in an organization. In addition, process capability determination identifies the capabilities of a process and the risks involved in it. The process improvement identifies the changes to be made in the software processes. The software capability determination motivates the organization to perform software process improvement.

Different approaches are used for assessing software process. These approaches are *SPICE* (*ISO/IEC15504*), *ISO 9001:2000*, *standard CMMI assessment method for process improvement*, *CMM-based appraisal for internal process improvement,* and *Bootstrap*.

SPICE (ISO/IEC15504)

SPICE (Software Process Improvement and Capability Determination) is a standard used for both process improvement and process capability determination. SPICE provides a framework for assessing the software process and is used by the organizations involved in planning, monitoring, developing, managing, and improving acquisitions. It is carried out in accordance with the International Organization for Standardization (ISO) and International Electro-technical Committee (IEC), which are used together and known as **ISO/IEC 15504**. The functions of SPICE (ISO/IEC 15504) are listed below.

NOTES

- To develop process-rating profiles instead of pass or fail criteria
- To consider the environment in which the assessed process operates
- To facilitate self assessment
- To ensure suitability for all applications and all sizes of organizations.

SPICE (ISO/IEC 15504) constitutes a set of documents that is used to guide goals and fundamental activities and grade the organization according to its level of capability. In addition, it determines the capability of a process in an organization, based on the results of the assessment of the processes and products. It also develops a maturity model for process improvement. This model is known as **SPICE reference model** as shown in Figure 2.2. It is applicable for all processes and comprises following six levels.

- **Not performed:** At this level, the processes are unable to accomplish the required outcomes. Thus, no identifiable products are created.
- **Performed informally:** At this level, the implemented process accomplishes the defined outcomes. It is not planned and tracked; rather it depends on individual knowledge and identifiable products.
- **Planned and tracked:** At this level, the defined process delivers products according to quality requirements within a specified time. The processes and products are verified according to the procedures, standards, and requirements.
- **Well-defined:** At this level, the processes based on software engineering principles which are capable of achieving defined outcomes are used.
- **Quantitatively controlled:** At this level, the performance measures, prediction capability and objective management are evaluated quantitatively. In addition, existing processes perform consistently within the defined limits to accomplish the desired outputs.
- **Continuously improved:** At this level, the existing processes adapt to meet future business goals. For continuous improvement, various kinds of statistical methods are used.

NOTES

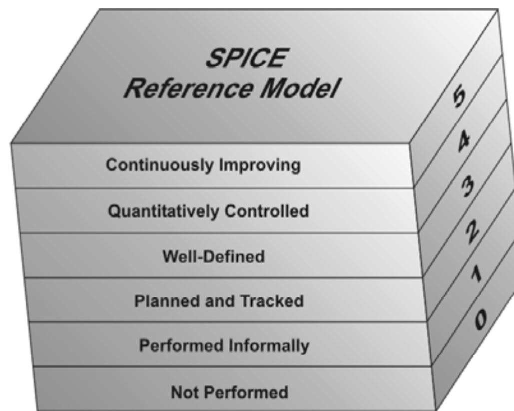


Fig. 2.2 *SPICE Reference Model*

ISO 9001:2000

ISO (International Organization for Standardization) established a standard known as ISO 9001:2000 to determine the requirements of quality management systems. A **quality management system** refers to the activities within an organization, which satisfies the quality related expectations of customers. Organizations ensure that they have a quality management system by demonstrating their conformance to the ISO 9001:2000 standard. The major advantage of this standard is that it achieves a better understanding and consistency of all quality practices throughout the organization. In addition, it strengthens the customer's confidence in the product. This standard follows a **plan-do-check-act** (**PDCA**) cycle (see Figure 2.3), which includes a set of activities that are listed below.

- **Plan:** Determines the processes and resources which are required to develop a quality product according to the user's satisfaction.
- **Do:** Performs activities according to the plan to create the desired product.
- **Check:** Measures whether the activities for establishing quality management according to the requirements are accomplished. In addition, it monitors the processes and takes corrective actions to improve them.
- **Act:** Initiates activities which constantly improve processes in the organization.

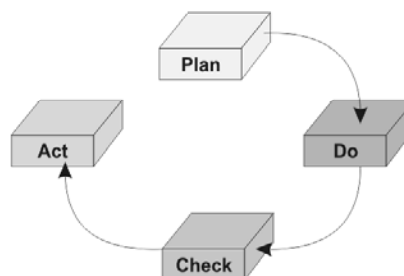


Fig. 2.3 *Cycle in ISO 9001:2000*

Note: The standard ISO 9001:2000 enhances the quality of a product by utilizing the processes for continual improvement of the system.

NOTES

Standard CMMI Appraisal Method for Process Improvement (SCAMPI)

An *appraisal* examines processes to determine their strengths and weaknesses in an organization. The appraisal used in an organization evaluates the internal processes (series of procedures occurring in the organization) for establishing or updating process improvement. In addition, appraisal includes measuring the process improvement progress by conducting audits of the processes. To conduct an appraisal, a scheme known as SCAMPI was developed by the **Software Engineering Institute** (**SEI**).

SCAMPI is used for process improvement by gaining insight into the process capability in the organization. The major advantage of SCAMPI is that it supports process improvement and establishes a consensus within the organization regarding areas where the process improvements are needed. The major characteristics of SCAMPI are listed in Table 2.1.

Table 2.1 *SCAMPI Characteristics*

Characteristic	*Description*
Accuracy	Ratings of the organization reflect its capability, which is used in comparison to other organizations. The result of appraisal indicates the strengths and weaknesses of the appraised organization.
Repeatability	Ratings and results of the appraisals are expected to be consistent with another appraisal conducted under comparable conditions (Another appraisal with identical scope will produce consistent result).
Cost/Resources effectiveness	The appraisal method is efficient as it takes into account the organizational investment in obtaining the appraisal results.

SCAMPI enables an organization to select and follow an approach for developing a plan for appraisal, which is appropriate for the requirements. These appraisal requirements in CMMI (ARC) comprise criteria for developing, defining, and utilizing the *appraisal methods* (namely, *Class A*, *Class B*, and *Class C*), which are based on **Capability Maturity Model Integration** (**CMMI**) as an improved framework. The objectives of SCAMPI are listed below.

- To identify strengths and weaknesses of existing processes in the organization
- To specify an integrated appraisal method for internal process improvement
- To act as a motivation for initiating and focusing on software process improvement.

SCAMPI is an appropriate tool for *benchmarking* within the organization. This process emphasizes a rigorous method that is capable of achieving high accuracy and reliability of appraisal results. The major characteristics of classes 'A', 'B', and 'C' are listed in Table 2.2. Class 'A' assessment method should satisfy all appraisal requirements of CMMI. The major characteristic of Class 'A'

assessment is a detailed assessment of process (es). With a thorough coverage, the strengths and weaknesses of processes are determined. SCAMPI is a Class 'A' assessment method for process improvement.

NOTES

Class 'B' appraisal method should comply with a subset of ARC requirements. As shown in Table 2.2, requirements of the Class 'A' method are optional for Class 'B'. Class 'B' assessment helps the organization to gain insight into its process capability. It focuses on areas that need improvement. It does not emphasize detailed coverage and is not efficient for level rating. These types of appraisals are considered efficient for initial assessment in organizations that have just started to use CMMI for process improvement activities. In addition, it is useful in providing a cost-effective measure for performing interim assessment and capability evaluations.

Table 2.2 *Characteristics of Appraisal Methods*

Characteristic	***Class A***	***Class B***	***Class C***
Amount of relative objectives gathered	High	Medium	Low
Ratings generated	Yes	No	No
Relative resources required	High	Medium	Low
Team size required for appraisal	Large	Medium	Small

Class 'C' appraisal methods are inexpensive and used for a short duration. In addition, they provide quick feedback to the result of the assessment. In other words, these appraisal methods are useful for periodic assessment of the projects.

Class 'B' and Class 'C' appraisals are useful for organizations that do not require generation of ratings. The primary reason for all appraisal methods should be to identify the strengths and weaknesses of the processes for their improvements.

CMM-based Appraisal for Internal Process Improvement (CBA IPI)

CBA-IPI tool is used in an organization to gain insight into the software development capability. For this, the strengths and weaknesses of the existing process are identified in order to prioritize software improvement plans and focus on software improvements, which are beneficial to the organization. The organization's software process capability is assessed by a group of individuals known as the **assessment team**, which generates findings and provides ratings according to the **CMM** (Capability Maturity Model). These findings are collected from questionnaires, document reviews and interviews with the managers of the organization. Thus, the primary goal of CBA IPI is to provide an actual picture of the existing processes in an organization. To achieve this, the assessment team performs the following functions.

- Provides data as a baseline to the organization in order to check its software capability
- Identifies issues that have an impact on the process improvement

- Provides sufficiently complete findings to the organization. These are used to guide the organization in planning and prioritizing future process improvement activities.

NOTES

For an assessment to be considered in CBA IPI, it should satisfy the minimum requirements concerning the assessment team, assessment plan, data collection, data validation, and reporting of the assessment results. These requirements are listed in Table 2.3.

Table 2.3 *Requirements in CBA IPI*

Requirement	***Description***
Assessment team	The size of the team should be minimum 4 and maximum 10 individuals. At least one team member should be from the organization.
Assessment plan	Contains goals for the assessment process. In addition, it has a schedule for assessment activities and identifies the resources required to perform the assessment. The assessment plan considers risks and the constraints, which are associated with the execution of the assessment.
Data collection	Data for process assessment is collected in the form of instruments, presentations, and documents.
Data validation	Data is validated using the rules of documentation.
Reporting of assessment	Provides summary of the whole assessment process, which presents both the strengths, and weaknesses of the processes.

The CBA IPI method is similar to SCAMPI as both are used for process assessment in an organization. However, differences do exist between the two approaches. These differences are listed in Table 2.4.

Table 2.4 *Differences between CBA IPI and SCAMPI*

Issue	***CBA IPI***	***SCAMPI***
Model based	Capability Maturity Model (CMM)	Capability Maturity Model Integration (CMMI)
Licensing	No	Yes
Authorization	Through training in assessor appraisal program	Through training in program
Cost	Less external cost due to internal resource usage	Costly due to model scope, appraisal complexity, and training
Performance	Less rigorous	More rigorous
Training authorized	Authorized lead assessors	Licensed and with lead appraisers

Note: The CMMI appraisal method provides a consistent rating for organizations to convert their appraisal ratings into a maturity level.

NOTES

Bootstrap

Bootstrap is an improvement on SEI approaches for process assessment and improvement and covers the requirements laid by ISO 9000. This approach evaluates and improves the quality of software development and management process of an organization. It defines a framework for assessing and promoting process improvement. The basic objectives of this approach are listed below.

- To support evaluation of the process capability
- To identify the strengths and weaknesses of the processes in the organization being assessed
- To support the accomplishment of goals in an organization by planning improved actions
- To increase the effectiveness of the processes while implementing standard requirements in the organization.

The main feature of the bootstrap approach is the assessment process, which leads to an improvement in the software development processes. During the assessment, the organizational processes are evaluated to define each process. Note that after the assessment is done, data is collected in a central database. In addition, it provides two questionnaires (forms containing a set of questions, which are distributed to people to gain statistical information): the first to gather data about the organization that develops the software and the second to gather data about the projects.

2.3 PERSONAL AND TEAM PROCESS MODELS

It is a structured software development process which is planned for software engineers to better understand and enhance their performance by bringing discipline to the way they develop software and tracing their predicted and actual development of the code.

In addition with the personal software process (PSP), the team software process (TSP) provides operational process framework which is designed to support teams of managers and engineers organize projects and produce software products that range in size from small projects of several thousand lines of code (KLOC) to very large projects greater than half a million lines of code. The TSP is planned to enhance the quality levels. It also improve productivity of a team's software development project, in order to help them better meet the cost and schedule commitments of developing a software system.

Check Your Progress

1. What is the aim of process assessment?
2. What is the main feature of boot strap?
3. What is the full form of SPICE?

2.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

NOTES

1. The aim of process assessment is to identify the areas for improvement and suggest a plan for making that improvement.
2. The main feature of the bootstrap approach is the assessment process, which leads to an improvement in the software development processes.
3. SPICE stands for Software Process Improvement and Capability Determination.

2.5 SUMMARY

- The process is assessed to evaluate methods, tools, and practices, which are used to develop and test the software. The aim of process assessment is to identify the areas for improvement and suggest a plan for making that improvement.
- SPICE provides a framework for assessing the software process and is used by the organizations involved in planning, monitoring, developing, managing, and improving acquisitions.
- A quality management system refers to the activities within an organization, which satisfies the quality related expectations of customers.
- An *appraisal* examines processes to determine their strengths and weaknesses in an organization. The appraisal used in an organization evaluates the internal processes (series of procedures occurring in the organization) for establishing or updating process improvement.
- The organization's software process capability is assessed by a group of individuals known as the **assessment team**, which generates findings and provides ratings according to the CMM (Capability Maturity Model).
- Team software process (TSP) provides a defined operational process framework that is designed to help teams of managers and engineers organize projects and produce software products that range in size from small projects of several thousand lines of code (KLOC) to very large projects greater than half a million lines of code.
- TSP was developed and piloted by Watts Humphrey in the late 1990s and the Technical Report for TSP sponsored by the U.S. Department of Defense was published in November 2000.
- The Personal Software Process (PSP) is a structured software development process that is intended to help software engineers better understand and improve their performance by bringing discipline to the way they develop software and tracking their predicted and actual development of the code.

- The PSP was created by Watts Humphrey to apply the underlying principles of the Software Engineering Institute's (SEI) Capability Maturity Model (CMM) to the software development practices of a single developer.

NOTES

2.6 KEY WORDS

- **SPICE (Software Process Improvement and Capability Determination):** It is a standard used for both process improvement and process capability determination.
- **Quality Management System:** It refers to the activities within an organization, which satisfies the quality related expectations of customers.

2.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Discuss the aim of process assessment.
2. Write the characteristics of SCAMPI method.
3. Discuss the differences between CBA IPI and SCAMPI.

Long Answer Questions

1. Explain the different approaches used for assessing software process.
2. Write a detailed note on CMMI method.

2.8 FURTHER READINGS

Pressman, Roger S. 1997. *Software Engineering, a Practitioner's Approach*. New Delhi: Tata McGraw-Hill.

Jalote, Pankaj. 1991. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

Schach, Stephen R. 2005. *Object Oriented and Classical Software Engineering*. New Delhi: Tata McGraw-Hill.

Somerville, Ian. 2001. *Software Engineering*. New Delhi: Pearson Education.

Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandriolli . 1991. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hill of India.

Jawadekar, Waman S. 2004. *Software Engineering: Principles and Practice*. New Delhi: Tata McGraw-Hill.

UNIT 3 PROCESS MODEL

NOTES

Structure

3.0 INTRODUCTION

A process model can be defined as a strategy (also known as software engineering paradigm), comprising process, methods, and tools layers as well as the general phases for developing the software. It provides a basis for controlling various activities required to develop and maintain the software. In addition, it helps the software development team in facilitating and understanding the activities involved in the project.

A process model for software engineering depends on the nature and application of the software project. Thus, it is essential to define process models for each software project. IEEE defines a process model as 'a framework containing the processes, activities, and tasks involved in the development, operation, and maintenance of a software product, spanning the life of the system from the definition of its requirements to the termination of its use.' A process model reflects the goals of software development such as developing a high quality product and meeting the schedule on time. In addition, it provides a flexible framework for enhancing the processes. Other advantages of the software process model are listed below.

- **Enables effective communication:** It enhances understanding and provides a specific basis for process execution.
- **Facilitates process reuse:** Process development is a time consuming and expensive activity. Thus, the software development team utilizes the existing processes for different projects.
- **Effective:** Since process models can be used again and again; reusable processes provide an effective means for implementing processes for software development.

NOTES

- **Facilitates process management:** Process models provide a framework for defining process status criteria and measures for software development. Thus, effective management is essential to provide a clear description of the plans for the software project.

3.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain the waterfall model
- Understand the incremental process model
- Explain the evolutionary process model and unified process

3.2 WATERFALL MODEL

In the waterfall model (also known as the **classical life cycle model**), the development of software proceeds linearly and sequentially from requirement analysis to design, coding, testing, integration, implementation, and maintenance. Thus, this model is also known as the **linear sequential model**.

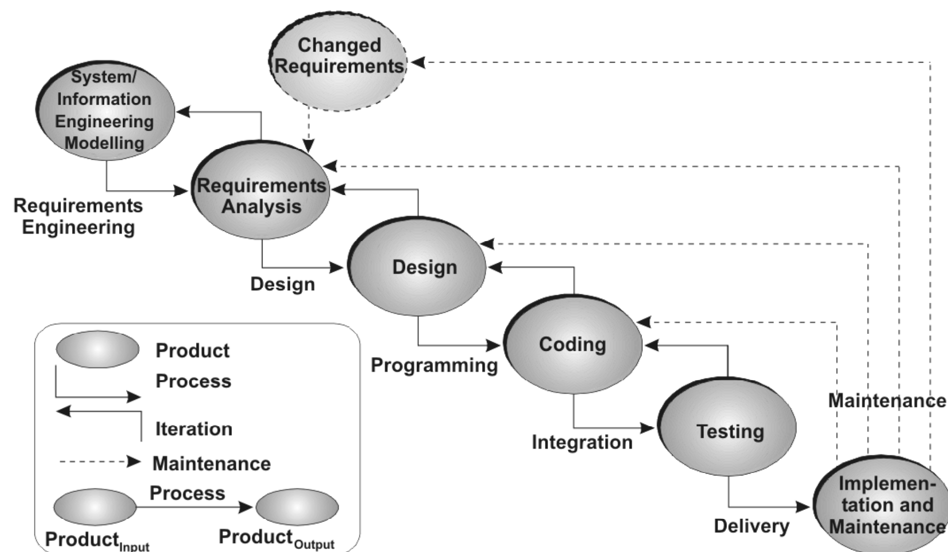


Fig. 3.1 The Waterfall Model

This model is simple to understand and represents processes which are easy to manage and measure. The waterfall model comprises different phases and each phase has its distinct goal. Figure 3.1 shows that after the completion of one phase, the development of software moves to the next phase. Each phase modifies the intermediate product to develop a new product as an output. The new product becomes the input of the next process. Table 3.1 lists the inputs and outputs of each phase of waterfall model.

NOTES

Table 3.1 Inputs and Outputs of each Phase of Waterfall Model

Input to Phase	*Process/ Phase*	*Output of the Phase*
Requirements defined requirements through communication	Requirements analysis	Software specification document
Software requirements specification document	Design	Design specification document
Design specification document	Coding	Executable software modules
Executable software modules	Testing	Integrated product
Integrated product	Implementation	Delivered software
Delivered software	Maintenance	Changed requirements

As stated earlier, the waterfall model comprises several phases, which are listed below.

- **System/information engineering modeling:** This phase establishes the requirements for all parts of the system. Software being a part of the larger system, a subset of these requirements is allocated to it. This system view is necessary when software interacts with other parts of the system including hardware, databases, and people. System engineering includes collecting requirements at the system level while information engineering includes collecting requirements at a level where all decisions regarding business strategies are taken.
- **Requirements analysis:** This phase focuses on the requirements of the software to be developed. It determines the processes that are to be incorporated during the development of the software. To specify the requirements, users' specifications should be clearly understood and their requirements be analyzed. This phase involves interaction between the users and the software engineers and produces a document known as **Software Requirements Specification** (**SRS**).
- **Design:** This phase determines the detailed process of developing the software after the requirements have been analyzed. It utilizes software requirements defined by the user and translates them into software representation. In this phase, the emphasis is on finding solutions to the problems defined in the requirements analysis phase. The software engineer is mainly concerned with the data structure, algorithmic detail and interface representations.
- **Coding:** This phase emphasizes translation of design into a programming language using the coding style and guidelines. The programs created should be easy to read and understand. All the programs written are documented according to the specification.

NOTES

- **Testing:** This phase ensures that the software is developed as per the user's requirements. Testing is done to check that the software is running efficiently and with minimum errors. It focuses on the internal logic and external functions of the software and ensures that all the statements have been exercised (tested). Note that testing is a multistage activity, which emphasizes verification and validation of the software.
- **Implementation and maintenance:** This phase delivers fully functioning operational software to the user. Once the software is accepted and deployed at the user's end, various changes occur due to changes in the external environment (these include upgrading a new operating system or addition of a new peripheral device). The changes also occur due to changing requirements of the user and changes occurring in the field of technology. This phase focuses on modifying software, correcting errors, and improving the performance of the software.

Various advantages and disadvantages associated with the waterfall model are listed in Table 3.2.

Table 3.2 *Advantages and Disadvantages of Waterfall Model*

Advantages	*Disadvantages*
• Relatively simple to understand. • Each phase of development proceeds sequentially. • Allows managerial control where a schedule with deadlines is set for each stage of development. • Helps in controlling schedules, budgets, and documentation.	• Requirements need to be specified before the development proceeds. • The changes of requirements in later phases of the waterfall model cannot be done. This implies that once the software enters the testing phase, it becomes difficult to incorporate changes at such a late phase. • No user involvement and working version of the software is available when the software is being developed. • Does not involve risk management. • Assumes that requirements are stable and are frozen across the project span.

3.3 EVOLUTIONARY PROCESS MODELS

Evolutionary software are the models that are iterative in nature. They models helps the software engineers to develop advance version of an available software, means initially a rapid version of the product is being developed. After that the product is developed to more accurate version with the help of the reviewers who review the product after each release and submit improvements. The main two evolutionary models are:

1. Incremental model
2. Spiral model

3.3.1 Incremental Process Model

NOTES

The incremental model (also known as **iterative enhancement model**) comprises the features of waterfall model in an iterative manner. The waterfall model performs each phase for developing complete software whereas the incremental model has phases similar to the linear sequential model and has an iterative nature of prototyping. As shown in Figure 3.2, during the implementation phase, the project is divided into small subsets known as **increments** that are implemented individually. This model comprises several phases where each phase produces an increment. These increments are identified in the beginning of the development process and the entire process from requirements gathering to delivery of the product is carried out for each increment.

The basic idea of this model is to start the process with requirements and iteratively enhance the requirements until the final software is implemented. In addition, as in prototyping, the increment provides feedback from the user specifying the requirements of the software. This approach is useful as it simplifies the software development process as implementation of smaller increments is easier than implementing the entire system.

As shown in Figure 3.2, each stage of incremental model adds some functionality to the product and passes it on to the next stage. The first increment is generally known as a **core product** and is used by the user for a detailed evaluation. This process results in creation of a plan for the next increment. This plan determines the modifications (features or functions) of the product in order to accomplish user requirements. The iteration process, which includes the delivery of the increments to the user, continues until the software is completely developed. The increments result in implementations, which are assessed in order to measure the progress of the product.

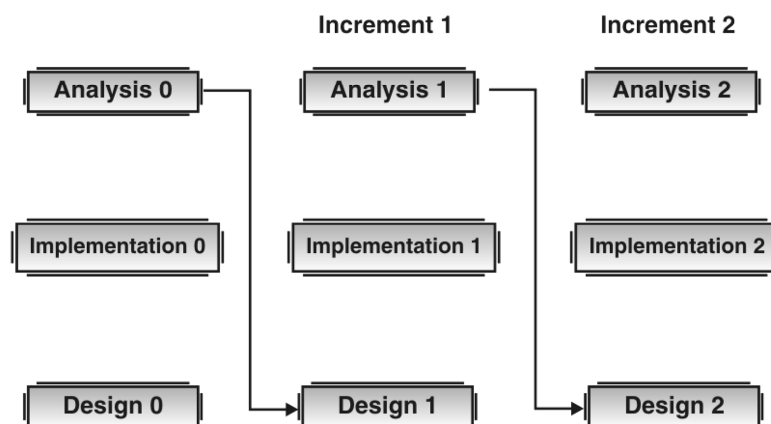


Fig. 3.2 *The Incremental Model*

NOTES

Various advantages and disadvantages associated with the incremental model are listed in Table 3.3.

Table 3.3 *Advantages and Disadvantages of Incremental Model*

Advantages	***Disadvantages***
• Avoids the problems resulting in risk-driven approach in the software. • Understanding increases through successive refinements. • Performs cost-benefit analysis before enhancing software with capabilities. • Incrementally grows in effective solution after every iteration. • Does not involve high complexity rate. • Early feedback is generated because implementation occurs rapidly for a small subset of the software.	• Requires planning at the management and technical level. • Becomes invalid when there is time constraint on the project schedule or when the users cannot accept the phased deliverables.

3.3.2 Spiral Model

In the 1980s, Boehm introduced a process model known as the spiral model. The spiral model comprises activities organized in a spiral, and has many cycles. This model combines the features of the prototyping model and waterfall model and is advantageous for large, complex, and expensive projects. It determines requirements problems in developing the prototypes. In addition, it guides and measures the need of risk management in each cycle of the spiral model. **IEEE** defines the spiral model as '*a model of the software development process in which the constituent activities, typical requirements analysis, preliminary and detailed design, coding, integration, and testing, are performed iteratively until the software is complete*.'

The objective of the spiral model is to emphasize management to evaluate and resolve risks in the software project. Different areas of risks in the software project are project overruns, changed requirements, loss of key project personnel, delay of necessary hardware, competition with other software developers and technological breakthroughs, which make the project obsolete. Figure 3.3 shows the spiral model.

Table 3.4 Advantages and Disadvantages of Prototyping Model

Advantages	Disadvantages
• Provides a working model to the user early in the process, enabling early assessment and increasing user's confidence. • The developer gains experience and insight by developing a prototype thereby resulting in better implementation of requirements. • The prototyping model serves to clarify requirements, which are not clear, hence reducing ambiguity and improving communication between the developers and users. • There is a great involvement of users in software development. Hence, the requirements of the users are met to the greatest extent. • Helps in reducing risks associated with the software.	• If the user is not satisfied by the developed prototype, then a new prototype is developed. This process goes on until a perfect prototype is developed. Thus, this model is time consuming and expensive. • The developer loses focus of the real purpose of prototype and hence, may compromise with the quality of the software. For example, developers may use some inefficient algorithms or inappropriate programming languages while developing the prototype. • Prototyping can lead to false expectations. For example, a situation may be created where the user believes that the development of the system is finished when it is not. • The primary goal of prototyping is speedy development, thus, the system design can suffer as it is developed in series without considering integration of all other components.

NOTES

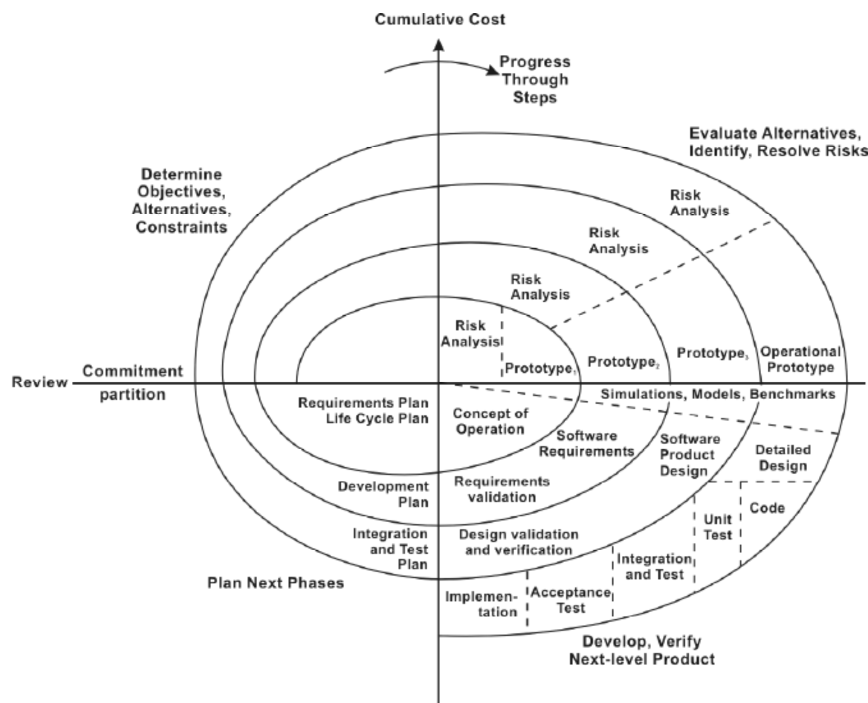


Fig. 3.3 *The Spiral Model*

The steps involved in the spiral model are listed below.

1. Each cycle of the first quadrant commences with **identifying the goals** for that cycle. In addition, it determines other alternatives, which are possible in accomplishing those goals.
2. The next step in the cycle **evaluates alternatives** based on objectives and constraints. This process identifies the areas of uncertainty and focuses

NOTES

on significant sources of the project risks. Risk signifies that there is a possibility that the objectives of the project cannot be accomplished. If so, the formulation of a cost-effective strategy for resolving risks is followed. Figure 3.3 shows the strategy, which includes *prototyping*, *simulation*, *benchmarking*, *administrating user*, *questionnaires,* and *risk resolution technique*.

3. The development of the software depends on remaining risks. The third quadrant **develops the final software** while considering the risks that can occur. Risk management considers the time and effort to be devoted to each project activity such as planning, configuration management, quality assurance, verification, and testing.
4. The last quadrant **plans the next step** and includes planning for the next prototype and thus, comprises the requirements plan, development plan, integration plan, and test plan.

One of the key features of the spiral model is that each cycle is completed by a review conducted by the individuals or users. This includes the review of all the intermediate products, which are developed during the cycles. In addition, it includes the plan for the next cycle and the resources required for that cycle.

The spiral model is similar to the waterfall model as software requirements are understood at the early stages in both the models. However, the major risks involved with developing the final software are resolved in the spiral model. When these issues are resolved, a detailed design of the software is developed. Note that processes in the waterfall model are followed by different cycles in the spiral model as shown in Figure 3.4.

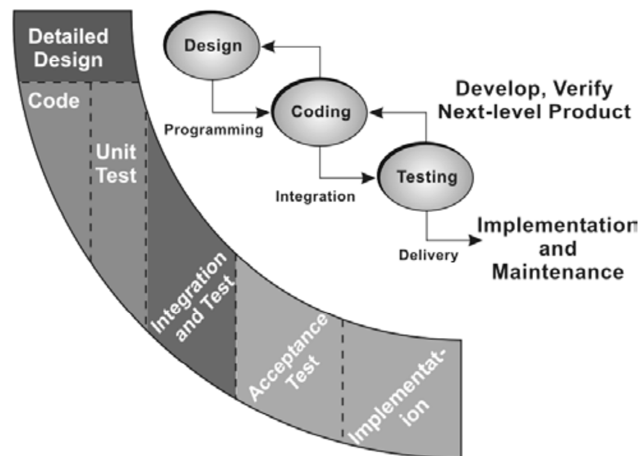


Fig. 3.4 *Spiral and Waterfall Models*

The spiral model is also similar to the prototyping model as one of the key features of prototyping is to develop a prototype until the user requirements are accomplished. The second step of the spiral model functions similarly. The prototype is developed to clearly understand and achieve the user requirements. If the user

NOTES

is not satisfied with the prototype, a new prototype known as **operational prototype** is developed.

Various advantages and disadvantages associated with the spiral model are listed in Table 3.5.

Table 3.5 *Advantages and Disadvantages of Spiral Model*

Advantages	***Disadvantages***
• Avoids the problems resulting in its risk-driven approach in the software.	• Assessment of project risks and resolution is not an easy task.
• Specifies a mechanism for software quality assurance activities.	• Difficult to estimate budget and schedule in the beginning as some of
• Is utilized by complex and dynamic projects. developed.	the analysis is not done until the design of the software is
• Re-evaluation after each step allows changes in user perspectives, technology advances, or financial perspectives.	
• Estimation of budget and schedule gets realistic as the work progresses.	

3.4 UNIFIED PROCESS

It is an iterative and incremental software development process framework. It is not only the process but an extensible framework that can be customized according to the organizational requirements. Rational unified process (RUP) is an extensively documented refinement of the Unified Process. The *Unified Process* name is also used to avoid potential issues of trademark infringement since *Rational Unified Process* and *RUP* are trademarks of IBM.

Various characteristics of unified process are:

1. Iterative and incremental
2. Architecture-centric
3. Risk-focused

Check Your Progress

1. Define waterfall model.
2. Define spiral model.
3. Write one of the key feature of spiral model.
4. What is unified process?

3.5 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

NOTES

1. The waterfall model development of software proceeds linearly and sequentially from requirement analysis to design, coding, testing, integration, implementation, and maintenance.
2. IEEE defines the spiral model as 'a model of the software development process in which the constituent activities, typical requirements analysis, preliminary and detailed design, coding, integration, and testing, are performed iteratively until the software is complete.'
3. One of the key features of the spiral model is that each cycle is completed by a review conducted by the individuals or users. This includes the review of all the intermediate products, which are developed during the cycles.
4. Unified Process is an iterative and incremental software development process framework.

3.6 SUMMARY

- The waterfall model (also known as the classical life cycle model), the development of software proceeds linearly and sequentially from requirement analysis to design, coding, testing, integration, implementation, and maintenance.
- The waterfall model performs each phase for developing complete software whereas the incremental model has phases similar to the linear sequential model and has an iterative nature of prototyping.
- Evolutionary software are the models that are iterative in nature.
- The incremental model (also known as iterative enhancement model) comprises the features of waterfall model in an iterative manner. The basic idea of this model is to start the process with requirements and iteratively enhance the requirements until the final software is implemented. In addition, as in prototyping, the increment provides feedback from the user specifying the requirements of the software. This approach is useful as it simplifies the software development process as implementation of smaller increments is easier than implementing the entire system.
- IEEE defines the spiral model as 'a model of the software development process in which the constituent activities, typical requirements analysis, preliminary and detailed design, coding, integration, and testing, are performed iteratively until the software is complete.'

- One of the key features of the spiral model is that each cycle is completed by a review conducted by the individuals or users. This includes the review of all the intermediate products, which are developed during the cycles.
- The Unified Process is an iterative and incremental software development process framework. It is not only the process but an extensible framework that can be customized according to the organizational requirements.
- Rational unified process (RUP) is an extensively documented refinement of the Unified Process.

3.7 KEY WORDS

- **Design:** This phase determines the detailed process of developing the software after the requirements have been analyzed. It utilizes software requirements defined by the user and translates them into software representation.
- **Coding:** This phase emphasizes translation of design into a programming language using the coding style and guidelines.
- **Testing:** This phase ensures that the software is developed as per the user's requirements. Testing is done to check that the software is running efficiently and with minimum errors.

3.8 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Discuss the inputs and outputs of each phase of waterfall model.
2. What are the advantages and disadvantages of waterfall model?
3. Write the advantages and disadvantages of iterative enhancement model.

Long Answer Questions

1. Explain the waterfall model for software development.
2. Explain the iterative enhancement model.
3. Write a note on evolutionary and unified process model.

3.9 FURTHER READINGS

NOTES

Pressman, Roger S. 1997. *Software Engineering, a Practitioner's Approach*. New Delhi: Tata McGraw-Hill.

Jalote, Pankaj. 1991. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

Schach, Stephen R. 2005. *Object Oriented and Classical Software Engineering*. New Delhi: Tata McGraw-Hill.

Somerville, Ian. 2001. *Software Engineering*. New Delhi: Pearson Education.

Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandriolli . 1991. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hill of India.

Jawadekar, Waman S. 2004. *Software Engineering: Principles and Practice*. New Delhi: Tata McGraw-Hill.

NOTES

BLOCK - II
REQUIREMENT ENGINEERING

UNIT 4 REQUIREMENT ENGINEERING–DESIGN AND CONSTRUCTION

Structure

4.0 INTRODUCTION

In the software development process, requirement phase is the first software engineering activity. This phase is a user-dominated phase and translates the ideas or views into a requirements document. Note that defining and documenting the user requirements in a concise and unambiguous manner is the first major step to achieve a high-quality product.

The requirement phase encompasses a set of tasks, which help to specify the impact of the software on the organization, customers' needs, and how users will interact with the developed software. The requirements are the basis of the system design. If requirements are not correct the end product will also contain errors.Note that requirements activity like all other software engineering activities shouldbe adapted to the needs of the process, the project, the product and the peopleinvolved in the activity. Also, the requirements should be specified at differentlevels of detail. This is because requirements are meant for people such as users,business managers, system engineers, and so on. For example, business managersare interested in knowing which features can be implemented within the allocatedbudget whereas end-users are interested in knowing how easy it is to use thefeatures of software.

NOTES

4.1 OBJECTIVES

After going through this unit, you will be able to:

- Define and understand software requirements
- Discuss the guidelines for expressing requirements
- Explain the different types of requirements
- Explain requirement engineering process
- Validate software requirements

4.2 REQUIREMENTS ENGINEERING TASK

Requirement is a condition or capability possessed by the software or system component in order to solve a real world problem. The problems can be to automate a part of a system, to correct shortcomings of an existing system, to control a device, and so on. **IEEE** defines requirement as '*(1) A condition or capability needed by a user to solve a problem or achieve an objective. (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents. (3) A documented representation of a condition or capability as in (1) or (2)*.'

Requirements describe how a system should act, appear or perform. For this, when users request for software, they provide an approximation of what the new system should be capable of doing. Requirements differ from one user to another and from one business process to another.

Guidelines for Expressing Requirements

The purpose of the requirements document is to provide a basis for the mutual understanding between the users and the designers of the initial definition of the software development life cycle (SDLC) including the requirements, operating environment and development plan.

The requirements document should include the overview, the proposed methods and procedures, a summary of improvements, a summary of impacts, security, privacy, internal control considerations, cost considerations, and alternatives. The requirements section should state the functions required in the software in quantitative and qualitative terms and how these functions will satisfy the performance objectives. The requirements document should also specify the performance requirements such as accuracy, validation, timing, and flexibility. Inputs, outputs, and data characteristics need to be explained. Finally, the requirements document needs to describe the operating environment and provide (or make reference to) a development plan.

There is no standard method to express and document requirements. Requirements can be stated efficiently by the experience of knowledgeable individuals, observing past requirements, and by following guidelines. Guidelines act as an efficient method of expressing requirements, which also provide a basis for software development, system testing, and user satisfaction. The guidelines that are commonly followed to document requirements are listed below.

- Sentences and paragraphs should be short and written in active voice. Also, proper grammar, spelling, and punctuation should be used.
- Conjunctions such as 'and' and 'or' should be avoided as they indicate the combination of several requirements in one requirement.
- Each requirement should be stated only once so that it does not create redundancy in the requirements specification document.

Types of Requirements

Requirements help to understand the behavior of a system, which is described by various tasks of the system. For example, some of the tasks of a system are to provide a response to input values, determine the state of data objects, and so on. Note that requirements are considered prior to the development of the software. The requirements, which are commonly considered, are classified into three categories, namely, *functional requirements*, *non-functional requirements*, and *domain requirements* (see Figure 4.1).

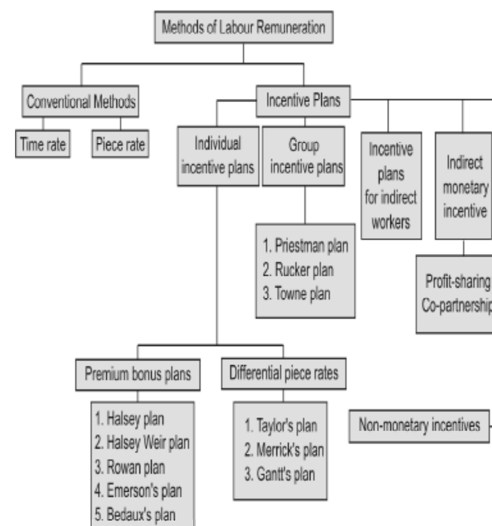


Fig. 4.1 *Types of Requirements*

Functional Requirements

IEEE defines functional requirements as '*a function that a system or component must be able to perform*.' These requirements describe the interaction of software with its environment and specify the inputs, outputs, external interfaces, and the functions that should be included in the software. Also, the services provided by

NOTES

functional requirements specify the procedure by which the software should react to particular inputs or behave in particular situations.

To understand functional requirements properly, let us consider the following example of an online banking system.

- The user of the bank should be able to search the desired services from the available ones.
- There should be appropriate documents for users to read. This implies that when a user wants to open an account in the bank, the forms must be available so that the user can open an account.
- After registration, the user should be provided with a unique acknowledgement number so that he can later be given an account number.

The above mentioned functional requirements describe the specific services provided by the online banking system. These requirements indicate user requirements and specify that functional requirements may be described at different levels of detail in an online banking system. With the help of these functional requirements, users can easily view, search and download registration forms and other information about the bank. On the other hand, if requirements are not stated properly, they are misinterpreted by software engineers and user requirements are not met.

The functional requirements should be complete and consistent. Completeness implies that all the user requirements are defined. Consistency implies that all requirements are specified clearly without any contradictory definition. Generally, it is observed that completeness and consistency cannot be achieved in large software or in a complex system due to the problems that arise while defining the functional requirements of these systems. The different needs of stakeholders also prevent the achievement of completeness and consistency. Due to these reasons, requirements may not be obvious when they are first specified and may further lead to inconsistencies in the requirements specification.

Non-functional Requirements

The non-functional requirements (also known as **quality requirements**) are related to system attributes such as reliability and response time. Non-functional requirements arise due to user requirements, budget constraints, organizational policies, and so on. These requirements are not related directly to any particular function provided by the system.

Non-functional requirements should be accomplished in software to make it perform efficiently. For example, if an aeroplane is unable to fulfill reliability requirements, it is not approved for safe operation. Similarly, if a real time control system is ineffective in accomplishing non-functional requirements, the control functions cannot operate correctly. Different types of non-functional requirements are shown in Figure 4.2.

The description of different types of non-functional requirements is listed below.

- **Product requirements:** These requirements specify how software product performs. Product requirements comprise the following.
    - o **Efficiency requirements:** Describe the extent to which the software makes optimal use of resources, the speed with which the system executes, and the memory it consumes for its operation. For example, the system should be able to operate at least three times faster than the existing system.
    - o **Reliability requirements:** Describe the acceptable failure rate of the software. For example, the software should be able to operate even if a hazard occurs.

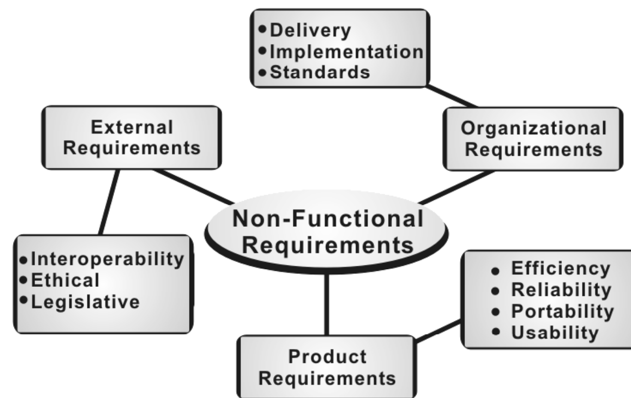


Fig. 4.2 *Types of Non-functional Requirements*

-
    - o **Portability requirements:** Describe the ease with which the software can be transferred from one platform to another. For example, it should be easy to port the software to a different operating system without the need to redesign the entire software.
    - o **Usability requirements:** Describe the ease with which users are able to operate the software. For example, the software should be able to provide access to functionality with fewer keystrokes and mouse clicks.
- **Organizational requirements:** These requirements are derived from the policies and procedures of an organization. Organizational requirements comprise the following.
    - o **Delivery requirements:** Specify when the software and its documentation are to be delivered to the user.
    - o **Implementation requirements:** Describe requirements such as programming language and design method.
    - o **Standards requirements:** Describe the process standards to be used during software development. For example, the software should be developed using standards specified by the ISO and IEEE standards.

NOTES

- **External requirements:** These requirements include all the requirements that affect the software or its development process externally. External requirements comprise the following.
  - o **Interoperability requirements:** Define the way in which different computer- based systems will interact with each other in one or more organizations.
  - o **Ethical requirements**: Specify the rules and regulations of the software so that they are acceptable to users.
  - o **Legislative requirements**: Ensure that the software operates within the legal jurisdiction. For example, pirated software should not be sold.

Non-functional requirements are difficult to verify. Hence, it is essential to write non-functional requirements quantitatively, so that they can be tested. For this, non-functional requirements metrics are used. These metrics are listed in Table 4.1.

Table 4.1 *Metrics for Non-functional Requirements*

Features	***Measures***
Speed	• Processed transaction/second • User/event response time • Screen refresh rate
Size	• Amount of memory (KB) • Number of RAM chips.
Ease of use	• Training time • Number of help windows
Reliability	• Mean time to failure (MTTF) • Portability of unavailability • Rate of failure occurrence
Robustness	• Time to restart after failure • Percentage of events causing failure • Probability of data corruption on failure
Portability	• Percentage of target-dependent statements • Number of target systems

Domain Requirements

Requirements which are derived from the application domain of the system instead from the needs of the users are known as **domain requirements**. These requirements may be new functional requirements or specify a method to perform some particular computations. In addition, these requirements include any constraint that may be present in the existing functional requirements. As domain requirements reflect the fundamentals of the application domain, it is important to understand these requirements. Also, if these requirements are not fulfilled, it may be difficult to make the system work as desired.

A system can include a number of domain requirements. For example, it may comprise a design constraint that describes the user interface, which is capable

of accessing all the databases used in a system. It is important for a development team to create databases and interface designs as per established standards. Similarly, the requirements of the user such as copyright restrictions and security mechanism for the files and documents used in the system are also domain requirements. When domain requirements are not expressed clearly, it can result in the following difficulties.

- **Problem of understandability:** When domain requirements are specified in the language of application domain (such as mathematical expressions), it becomes difficult for software engineers to understand them.
- **Problem of implicitness:** When domain experts understand the domain requirements but do not express these requirements clearly, it may create a problem (due to incomplete information) for the development team to understand and implement the requirements in the system.

Note: Information about requirements is stored in a database, which helps the software development team to understand user requirements and develop the software according to those requirements.

4.2.1 Requirements Engineering Process

This process is a series of activities that are performed in the requirements phase to express requirements in the Software Requirements Specification (SRS) document. It focuses on understanding the requirements and its type so that an appropriate technique is determined to carry out the **Requirements Engineering** (**RE**) **process**. The new software developed after collecting requirements either replaces the existing software or enhances its features and functionality. For example, the payment mode of the existing software can be changed from payment through hand-written cheques to electronic payment of bills.

In Figure 4.3, an RE process is shown, which comprises various steps including *feasibility study*, *requirements elicitation*, *requirements analysis*, *requirements specification*, *requirements validation*, and *requirements management*.

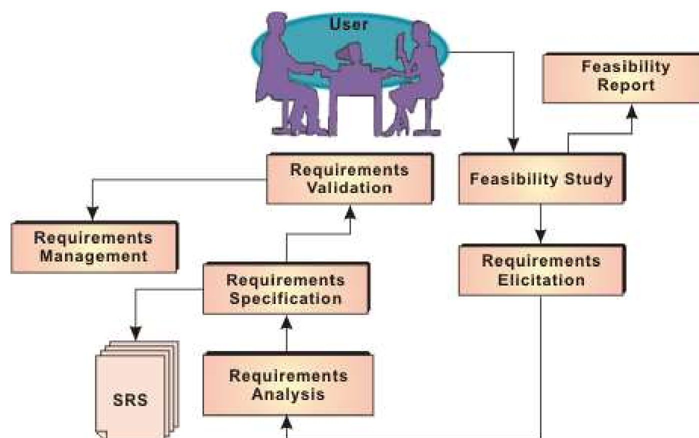


Fig. 4.3 *Requirements Engineering Process*

NOTES

The requirements engineering process begins with feasibility study of the requirements. Then requirements elicitation is performed, which focuses on gathering user requirements. After the requirements are gathered, an analysis is performed, which further leads to requirements specification. The output of this is stored in the form of software requirements specification document. Next, the requirements are checked for their completeness and correctness in requirements validation. Last of all, to understand and control changes to system requirements, requirements management is performed.

Feasibility Study

Feasibility is defined as the practical extent to which a project can be performed successfully. To evaluate feasibility, a feasibility study is performed, which determines whether the solution considered to accomplish the requirements is practical and workable in the software. Information such as resource availability, cost estimation for software development, benefits of the software to the organization after it is developed and cost to be incurred on its maintenance are considered during the feasibility study. The objective of the feasibility study is to establish the reasons for developing the software that is acceptable to users, adaptable to change and conformable to established standards. Various other objectives of feasibility study are listed below.

- To analyze whether the software will meet organizational requirements
- To determine whether the software can be implemented using the current technology and within the specified budget and schedule
- To determine whether the software can be integrated with other existing software.

Types of Feasibility

Various types of feasibility that are commonly considered include *technical feasibility, operational feasibility*, and *economic feasibility* (see Figure 4.4).

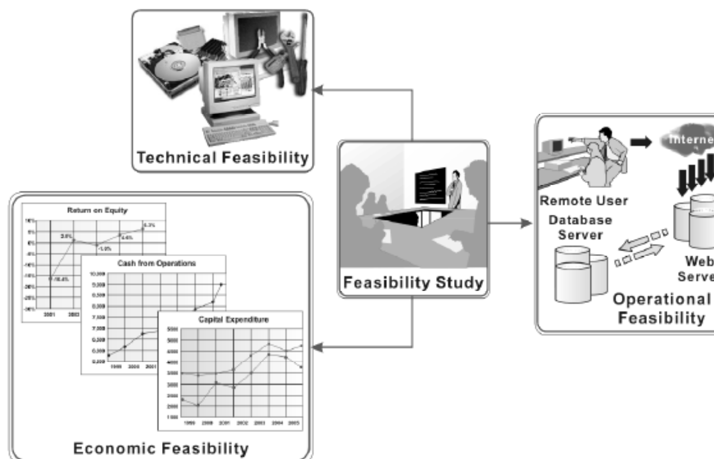


Fig. 4.4 *Types of Feasibility*

Technical Feasibility

Technical feasibility assesses the current resources (such as hardware and software) and technology, which are required to accomplish user requirements in the software within the allocated time and budget. For this, the software development team ascertains whether the current resources and technology can be upgraded or added in the software to accomplish specified user requirements. Technical feasibility also performs the following tasks.

- Analyzes the technical skills and capabilities of the software development team members
- Determines whether the relevant technology is stable and established
- Ascertains that the technology chosen for software development has a large number of users so that they can be consulted when problems arise or improvements are required.

Operational Feasibility

Operational feasibility assesses the extent to which the required software performs a series of steps to solve business problems and user requirements. This feasibility is dependent on human resources (software development team) and involves visualizing whether the software will operate after it is developed and be operative once it is installed. Operational feasibility also performs the following tasks.

- Determines whether the problems anticipated in user requirements are of high priority
- Determines whether the solution suggested by the software development team is acceptable
- Analyzes whether users will adapt to a new software
- Determines whether the organization is satisfied by the alternative solutions proposed by the software development team.

Economic Feasibility

Economic feasibility determines whether the required software is capable of generating financial gains for an organization. It involves the cost incurred on the software development team, estimated cost of hardware and software, cost of performing feasibility study, and so on. For this, it is essential to consider expenses made on purchases (such as hardware purchase) and activities required to carry out software development. In addition, it is necessary to consider the benefits that can be achieved by developing the software. Software is said to be economically feasible if it focuses on the issues listed below.

- Cost incurred on software development to produce long-term gains for an organization

- Cost required to conduct full software investigation (such as requirements elicitation and requirements analysis)
- Cost of hardware, software, development team, and training.

NOTES

Feasibility Study Process

Feasibility study comprises the following steps.

- **Information assessment:** Identifies information about whether the system helps in achieving the objectives of the organization. It also verifies that the system can be implemented using new technology and within the budget and whether the system can be integrated with the existing system.
- **Information collection:** Specifies the sources from where information about software can be obtained. Generally, these sources include users (who will operate the software), organization (where the software will be used), and the software development team (which understands user requirements and knows how to fulfill them in software).
- **Report writing:** Uses a feasibility report, which is the conclusion of the feasibility study by the software development team. It includes the recommendations whether the software development should continue. This report may also include information about changes in the software scope, budget, and schedule and suggestions of any requirements in the system.

1.0 General Information
1.1 Purpose
1.2 Scope
1.3 System Overview
1.4 Project References
1.5 Acronyms and Abbreviations
1.6 Points of Contacts
1.6.1 Information
1.6.2 Co-ordination
2.0 Management Summary
2.1 Environment
2.1.1 Organizations Involved
2.1.2 Input/Output
2.1.3 Processing
2.1.4 Security
2.1.5 System Interaction
2.1.6 Physical Environment
2.2 Current Functional Procedures
2.3 Functional Objectives
2.4 Performance Objectives
2.5 Assumptions and Constraints
2.6 Methodology
2.7 Evaluation Criteria
2.8 Recommendations

3.0 Proposed System
3.1 Description of a Proposed System
3.2 Improvements
3.3 Time and Resource Costs
3.4 Impacts
3.4.1 Equipment Impacts
3.4.2 Software Impacts
3.4.3 Organizational Impacts
3.4.4 Operational Impacts
3.4.5 Developmental Impacts
3.4.6 Site or Facility Impacts
3.4.7 Security and Privacy Impacts
3.5 Rationale for Recommendations
4.0 Alternative System
4.1 Description

Fig. 4.5 *Feasibility Study Plan*

Figure 4.5 shows the feasibility study plan, which comprises the following sections.

- **General information:** Describes the purpose and scope of feasibility study. It also describes *system overview*, *project references*, *acronyms and abbreviations*, and *points of contact* to be used. **System overview** provides description about the name of the organization responsible for the software development, system name or title, system category, operational status, and so on. **Project references** provide a list of the references used to prepare this document such as documents relating to the project or

NOTES

previously developed documents that are related to the project. **Acronyms and abbreviations** provide a list of the terms that are used in this document along with their meanings. **Points of contact** provide a list of points of organizational contact with users for information and coordination. For example, users require assistance to solve problems (such as troubleshooting) and collect information such as contact number, e-mail address, and so on.

- **Management summary:** Provides the following information.
  - o **Environment:** Identifies the individuals responsible for software development. It provides information about input and output requirements, processing requirements of the software and the interaction of the software with other software. It also identifies system security requirements and the system's processing requirements
  - o **Current functional procedures**: Describes the current functional procedures of the existing system, whether automated or manual. It also includes the data-flow of the current system and the number of team members required to operate and maintain the software.
  - o **Functional objective:** Provides information about functions of the system such as new services, increased capacity, and so on.
  - o **Performance objective:** Provides information about performance objectives such as reduced staff and equipment costs, increased processing speeds of software, and improved controls.
  - o **Assumptions and constraints:** Provides information about assumptions and constraints such as operational life of the proposed software, financial constraints, changing hardware, software and operating environment, and availability of information and sources.
  - o **Methodology:** Describes the methods that are applied to evaluate the proposed software in order to reach a feasible alternative. These methods include survey, modeling, benchmarking, etc.
  - o **Evaluation criteria:** Identifies criteria such as cost, priority, development time, and ease of system use, which are applicable for the development process to determine the most suitable system option.
  - o **Recommendation:** Describes a recommendation for the proposed system. This includes the delays and acceptable risks.
- **Proposed software:** Describes the overall concept of the system as well as the procedure to be used to meet user requirements. In addition, it provides information about improvements, time and resource costs, and impacts. Improvements are performed to enhance the functionality and performance of the existing software. Time and resource costs include the costs associated with software development from its requirements to its maintenance and staff training. Impacts describe the possibility of future happenings and include various types of impacts as listed below.

NOTES

- o **Equipment impacts:** Determine new equipment requirements and changes to be made in the currently available equipment requirements.
- o **Software impacts:** Specify any additions or modifications required in the existing software and supporting software to adapt to the proposed software.
- o **Organizational impacts:** Describe any changes in organization, staff and skills requirement.
- o **Operational impacts:** Describe effects on operations such as user-operating procedures, data processing, data entry procedures, and so on.
- o **Developmental impacts:** Specify developmental impacts such as resources required to develop databases, resources required to develop and test the software, and specific activities to be performed by users during software development.
- o **Security impacts:** Describe security factors that may influence the development, design, and continued operation of the proposed software.

- **Alternative systems:** Provide description of alternative systems, which are considered in a feasibility study. This also describes the reasons for choosing a particular alternative system to develop the proposed software and the reason for rejecting alternative systems.

Note: Economic feasibility uses several methods to perform cost-benefit analysis such as payback analysis, return on investment (ROI) and present value analysis.

Check Your Progress

1. Define requirement.
2. What are the different types of requirements?

4.3 REQUIREMENTS VALIDATION

The development of software begins once the requirements document is 'ready'. One of the objectives of this document is to check whether the delivered software system is acceptable. For this, it is necessary to ensure that the requirements specification contains no errors and that it specifies the user's requirements correctly. Also, errors present in the SRS will adversely affect the cost if they are detected later in the development process or when the software is delivered to the user. Hence, it is desirable to detect errors in the requirements before the design and development of the software begins. To check all the issues related to requirements, requirements validation is performed.

In the validation phase, the work products produced as a consequence of requirements engineering are examined for consistency, omissions, and ambiguity. The basic objective is to ensure that the SRS reflects the actual requirements accurately and clearly. Other objectives of the requirements document are listed below.

- To certify that the SRS contains an acceptable description of the system to be implemented
- To ensure that the actual requirements of the system are reflected in the SRS
- To check the requirements document for completeness, accuracy, consistency, requirement conflict, conformance to standards and technical errors.

Requirements validation is similar to requirements analysis as both processes review the *gathered requirements*. Requirements validation studies the 'final draft' of the requirements document while requirements analysis studies the 'raw requirements' from the system stakeholders (users). Requirements validation and requirements analysis can be summarized as follows:

- **Requirements validation:** Have we got the requirements right?
- **Requirements analysis:** Have we got the right requirements?

In Figure 4.6, various inputs such as *requirements document*, *organizational knowledge*, and *organizational standards* are shown. The **requirements document** should be formulated and organized according to the standards of the organization. The **organizational knowledge** is used to estimate the realism of the requirements of the system. The **organizational standards** are the specified standards followed by the organization according to which the system is to be developed.



Fig. 4.6 *Requirements Validation*

The output of requirements validation is a *list of problems* and *agreed actions* of the problems. The **lists of problems** indicate the problems encountered in the requirements document of the requirements validation process. The **agreed actions** is a list that displays the actions to be performed to resolve the problems depicted in the problem list.

NOTES

Requirements Review

Requirements validation determines whether the requirements are substantial to design the system. The problems encountered during requirements validation are listed below.

- Unclear stated requirements
- Conflicting requirements are not detected during requirements analysis
- Errors in the requirements elicitation and analysis
- Lack of conformance to quality standards.

To avoid the problems stated above, a **requirements review** is conducted, which consists of a review team that performs a systematic analysis of the requirements. The review team consists of software engineers, users, and other stakeholders who examine the specification to ensure that the problems associated with consistency, omissions, and errors are detected and corrected. In addition, the review team checks whether the work products produced during the requirements phase conform to the standards specified for the process, project, and the product.

At the review meeting, each participant goes over the requirements before the meeting starts and marks the items which are dubious or need clarification. Checklists are often used for identifying such items. Checklists ensure that no source of errors, whether major or minor, is overlooked by the reviewers. A 'good' checklist consists of the following.

- Is the initial state of the system defined?
- Is there a conflict between one requirement and the other?
- Are all requirements specified at the appropriate level of abstraction?
- Is the requirement necessary or does it represent an add-on feature that may not be essentially implemented?
- Is the requirement bounded and has a clear defined meaning?
- Is each requirement feasible in the technical environment where the product or system is to be used?
- Is testing possible once the requirement is implemented?
- Are requirements associated with performance, behavior, and operational characteristics clearly stated?
- Are requirements patterns used to simplify the requirements model?
- Are the requirements consistent with the overall objective specified for the system/product?
- Have all hardware resources been defined?
- Is the provision for possible future modifications specified?

NOTES

- Are functions included as desired by the user (and stakeholder)?
- Can the requirements be implemented in the available budget and technology?
- Are the resources of requirements or any system model (created) stated clearly?

The checklists ensure that the requirements reflect users' needs and provide groundwork for design. Using the checklists, the participants specify the list of potential errors they have uncovered. Lastly, the requirements analyst either agrees to the presence of errors or states that no errors exist.

Other Requirements Validation Techniques

A number of other requirements validation techniques are used either individually or in conjunction with other techniques to check the entire system or parts of the system. The selection of the validation technique depends on the appropriateness and the size of the system to be developed. Some of these techniques are listed below.

- **Test case generation:** The requirements specified in the SRS document should be testable. The test in the validation process can reveal problems in the requirement. In some cases test becomes difficult to design, which implies that the requirement is difficult to implement and requires improvement.
- **Automated consistency analysis:** If the requirements are expressed in the form of structured or formal notations, then CASE tools can be used to check the consistency of the system. A requirements database is created using a CASE tool that checks the entire requirements in the database using rules of method or notation. The report of all inconsistencies is identified and managed.
- **Prototyping:** Prototyping is normally used for validating and eliciting new requirements of the system. This helps to interpret assumptions and provide an appropriate feedback about the requirements to the user. For example, if users have approved a prototype, which consists of graphical user interface, then the user interface can be considered validated.

Check Your Progress

3. What are domain requirements?
4. Define feasibility study.
5. What is the use of prototyping?

4.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

NOTES

1. Requirement is a condition or capability possessed by the software or system component in order to solve a real world problem.
2. The requirements, which are commonly considered, are classified into three categories, namely, functional requirements, non-functional requirements, and domain requirements.
3. Requirements which are derived from the application domain of the system instead from the needs of the users are known as domain requirements.
4. Feasibility is defined as the practical extent to which a project can be performed successfully.
5. Prototyping is normally used for validating and eliciting new requirements of the system.

4.5 SUMMARY

- Requirement is a condition or capability possessed by the software or system component in order to solve a real world problem.
- The non-functional requirements (also known as quality requirements) are related to system attributes such as reliability and response time.
- Requirements which are derived from the application domain of the system instead from the needs of the users are known as domain requirements.
- Operational feasibility assesses the extent to which the required software performs a series of steps to solve business problems and user requirements.
- The development of software begins once the requirements document is ‘ready’.
- The requirements document should be formulated and organized according to the standards of the organization.
- Economic feasibility determines whether the required software is capable of generating financial gains for an organization. It involves the cost incurred on the software development team, estimated cost of hardware and software, cost of performing feasibility study.

4.6 KEY WORDS

- **Requirement:** A condition or a capability possessed by software or system component in order to solve a real world problem.

NOTES

- **Functional Requirements:** The requirements which describe the functionality or services that software should provide.
- **Non-functional Requirements** (also known as **quality requirements**): The requirements related to system attributes such as reliability and response time.
- **Domain Requirements:** The requirements which are derived from the application domain of a system, instead from the needs of the users.

4.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Define requirement.
2. Discuss the guidelines for expressing requirements.
3. What are the different types of requirements?

Long Answer Questions

1. Differentiate between functional and non-functional requirements.
2. Explain the requirement engineering process.
3. What is feasibility study? Explain its types.
4. What are the requirement validation techniques? Explain.

4.8 FURTHER READINGS

Pressman, Roger S. 1997. *Software Engineering, a Practitioner's Approach*. New Delhi: Tata McGraw-Hill.

Jalote, Pankaj. 1991. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

Schach, Stephen R. 2005. *Object Oriented and Classical Software Engineering*. New Delhi: Tata McGraw-Hill.

Somerville, Ian. 2001. *Software Engineering*. New Delhi: Pearson Education.

Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandriolli . 1991. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hill of India.

Jawadekar, Waman S. 2004. *Software Engineering: Principles and Practice*. New Delhi: Tata McGraw-Hill.

UNIT 5 BUILDING THE ANALYSIS MODEL

NOTES

Structure

5.0 INTRODUCTION

In this unit, you will learn about the requirement elicitation, analysis and data modeling concepts. Requirement elicitation is a process of collecting information about software requirements from different individuals. Requirements analysis helps to understand, interpret, classify, and organize the software requirements in order to assess the feasibility, completeness, and consistency of the requirements. You will also learn about the object oriented analysis which is used to describe the system requirements using prototypes.

5.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain various elicitation techniques
- Build the requirement model
- Understand the elements of requirement model
- Explain the data modeling concept
- Understand object-oriented analysis

5.2 REQUIREMENTS ELICITATION AND ANALYSIS

Requirements elicitation (also known as **requirements capture** and **requirements acquisition**) is a process of collecting information about software requirements from different individuals, such as users and other stakeholders. Stakeholders are

individuals who are affected by the system, directly or indirectly. They include project mangers, marketing personnel, consultants, software engineers, maintenance engineers, and user.

Various issues may arise during requirements elicitation and may cause difficulties in understanding the software requirements. Some of the problems are:

- **Problems of Scope**: This problem arises when the boundary of software (that is, scope) is not defined properly. Due to this, it becomes difficult to identify objectives as well as functions and features to be accomplished in the software.
- **Problems of Understanding**: This problem arises when users are not certain about their requirements and thus are unable to express what they require in a software and which requirements are feasible. This problem also arises when users have no or little knowledge of the problem domain and are unable to understand the limitations of the computing environment of a software.
- **Problems of Volatility**: This problem arises when requirements change over time.

Requirements elicitation uses *elicitation techniques*, which facilitate a software engineer to understand user requirements and software requirements needed to develop the proposed software.

Elicitation techniques

Various elicitation techniques are used to identify the problem, determine its solution, and identify different approaches for the solution. These techniques also help the stakeholders to clearly express their requirements by stating all the important information. The commonly followed elicitation techniques are:

- **Interviews**: These are conventional ways for eliciting requirements, which help software engineers, users, and software development team to understand the problem and suggest solutions for it. For this, the software engineer interviews the users with a series of questions. When an interview is conducted, rules are established for users and other stakeholders. In addition, an agenda is prepared before conducting interviews, which includes the important points (related to software) to be discussed among users and other stakeholders. An effective interviews should have the following characteristics:
    - o Individuals involved in interviews should be able to accept new ideas. Also, they should focus on listening to the views of stakeholders related to requirements and avoid biased views.
    - o Interviews should be conducted in defined context to requirements rather than in general terms. For this, users should start with a question or a *requirements proposal*.

NOTES

- **Scenarios**: These are descriptions of a sequence of events, which help to determine possible future outcome before a software is developed or implemented. Scenarios are used to *test* whether the software will accomplish user requirements or not. Also, scenarios help to provide a framework for questions to software engineer about users' tasks. These questions are asked from users about future conditions (what-if) and procedure (how) in which they think tasks can be completed. Generally, a scenario includes the following information:
  - o Description of what users expect when the scenario starts.
  - o Description of how to handle the situation when a software is not functioning properly.
  - o Description of the state of the software when the scenario ends.
- **Prototypes**: Prototypes help to clarify unclear requirements. Like scenarios, prototypes also help users to understand the information they need to provide to software development team.
- **Quality Function Deployment (QFD)**: This deployment translates user requirements into technical requirements for the software. For this, QFD facilitates development team to understand what is valuable to users so that quality can be maintained throughout software development. QFD identifies some of the common user requirements, namely:
  - o **General requirements**: These requirements describe the system objectives, which are determined by various requirements elicitation techniques. Examples of general requirements are graphical displays requested by users, specific software functions, and so on.
  - o **Expected requirements**: These requirements are implicit to software, as users consider them to be fundamental requirements, which will be accomplished in the software and hence do not express them. Examples of expected requirements are ease of software installation, ease of software and user interaction, and so on.
  - o **Unexpected requirements**: These requirements specify the requirements that are beyond user expectations. These requirements are not requested by the user but if added to the software, they become an additional feature of the software. An example of unexpected requirements is to have word processing software with additional capabilities, such as page layout capabilities along with the earlier features.

RequirementS analysis

IEEE defines requirements analysis as *'(1) the process of studying user needs to arrive at a definition of a system, hardware, or software requirements. (2) the process of studying and refining system, hardware, or software requirements*.'

Requirements analysis helps to understand, interpret, classify, and organize the software requirements in order to assess the feasibility, completeness, and consistency of the requirements. Various other tasks performed using requirements analysis are listed below.

- To detect and resolve conflicts that arises due to unclear and unspecified requirements
- To determine operational characteristics of software and how they interact with the environment
- To understand the problem for which the software is to be developed
- To develop an analysis model to analyze the requirements in the software.

Building the Requirements Model

Software engineers perform requirements analysis and create a **requirements model** (also known as **analysis model**) to provide information of 'what' software should do instead of 'how' to fulfill the requirements in software. This model emphasizes information such as the functions that software should perform, behavior it should exhibit, and constraints that are applied on the software. This model also determines the relationship of one component with other components. The clear and complete requirements specified in the analysis model help the software development team to develop the software according to those requirements. An analysis model is created to help the development team to assess the quality of the software when it is developed. An analysis model helps to define a set of requirements that can be validated when the software is developed.

Let us consider an example of constructing a study room, where the user knows the dimensions of the room, the location of doors and windows, and the available wall space. Before constructing the study room, he provides information about flooring, wallpaper, and so on to the constructor. This information helps the constructor to analyze the requirements and prepare an analysis model that describes the requirements. This model also describes what needs to be done to accomplish those requirements. Similarly, an analysis model created for the software facilitates the software development team to understand what is required in the software and then develop it.

In Figure 5.1, the analysis model connects the system description and design model. System description provides information about the entire functionality of the system, which is achieved by implementing the software, hardware and data. In addition, the analysis model specifies the software design in the form of a design model, which provides information about the software's architecture, user interface, and component level structure.

NOTES

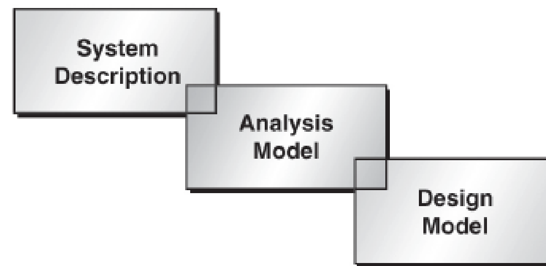


Fig. 5.1 *Analysis Model as Connector*

The guidelines followed while creating an analysis model are listed below.

- The model should concentrate on requirements in the problem domain that are to be accomplished. However, it should not describe the procedure to accomplish requirements in the system.
- Every element of the analysis model should help in understanding the software requirements. This model should also describe the information domain, function and behavior of the system.
- The analysis model should be useful to all stakeholders because every stakeholder uses this model in his own manner. For example, business stakeholders use this model to validate requirements whereas software designers view this model as a basis for design.
- The analysis model should be as simple as possible. For this, additional diagrams that depict no new or unnecessary information should be avoided. Also, abbreviations and acronyms should be used instead of complete notations.

Elements of requirements model

Requirements for a computer-based system can be represented in different ways. Some people believe that it is better to chose one mode of representation whereas others believe that requirements model should be described using different modes as different modes of representation allow considering the requirements from different perspectives.

Each requirements model consists of some set of elements which are specific to that model only. However, there exists a set of generic elements, which are common to most of the analysis models. These elements are as follows:

- **Scenario-based elements:** Scenario based elements form the first part of the analysis model that is developed. These elements act as input for the creation of other modeling elements. Scenario-based modeling gives a high level idea of the system from the user's point of view. The basic use cases, use-case diagrams and activity diagrams are the examples of scenario-based elements.

- **Class-based elements:** A class is a collection of things that have similar attributes and common behavior. It forms the basis for the objet-oriented software development. Thus, class-based elements are basically used in OO software. They give a static view of the system and how the different parts (or classes) are related. UML class diagrams, analysis packages, CRC models and collaboration diagrams are the example of class-based elements. The UML class diagram lists the attributes of a class and the operations that can be applied to modify these attributes.
- **Behavioral elements**: As the name suggests, the behavioral elements depict the behavior of the system. They describe how the system responds to external stimuli. State diagrams and activity diagrams are the examples of behavioral elements. The state transition diagram depicts the various possible states of a system and the events that cause transition from state to another. It also lists the actions to be taken in response to a particular event.
- **Flow-oriented elements:** A computer-based system accepts input in different forms, transform them using functions and produces output in different forms. The flow-oriented elements depict how the information flows throughout the system. Data-flow diagrams (DFD) and control-flow diagrams are the examples of flow-oriented elements.

Requirements Modeling Approaches

Requirements modeling is a technical representation of the system and there exist a variety of approaches for building the requirements models. Two common approaches include *structured analysis* and *object-oriented modeling*. Each of these describes a different manner to represent the data, functional, and behavioral information of the system.

Structured analysis

Structured analysis is a top-down approach, which focuses on refining the problem with the help of functions performed in the problem domain and data produced by these functions. This approach facilitates the software engineer to determine the information received during analysis and to organize the information in order to avoid the complexity of the problem. The purpose of structured analysis is to provide a graphical representation to develop new software or enhance the existing software.

Object-oriented modeling will be discussed later in this unit.

Check Your Progress

1. What is requirement elicitation?
2. Define prototypes.

5.3 DATA MODELING CONCEPTS

NOTES

The data model depicts three interrelated aspects of the system: *data objects* that are to be processed, *attributes* that characterize the data objects, and the *relationship* that links different data objects. Entity relationship (ER) model/diagram is the most commonly used model for data modeling.

Entity relationship diagram

IEEE defines ER diagram as '*a diagram that depicts a set of real-world entities and the logical relationships among them*'. This diagram depicts entities, the relationships between them, and the attributes pictorially in order to provide a high-level description of conceptual data models. An ER diagram is used in different phases of software development.

Once an ER diagram is created, the information represented by it is stored in the database. Note that the information depicted in an ER diagram is independent of the type of database and can later be used to create database of any kind, such as relational database, network database, or hierarchical database. An ER diagram comprises *data objects and entities*, *data attributes*, *relationships*, and *cardinality and modality*.

Data objects and entities

Data object is a representation of composite information used by software. Composite information refers to different features or attributes of a data object and this object can be in any of the following forms:

- **External entity**: describes the data that produces or accepts information. For example, a report.
- **Occurrence**: describes an action of a process. For example, a telephone call.
- **Event**: describes a happening that occurs at a specific place or time. For example, an alarm.
- **Role**: describes the actions or activities assigned to an individual or object. For example, a systems analyst.
- **Place**: describes the location of objects or storage area. For example, a wardrobe.
- **Structure**: describes the arrangement and composition of objects. For example, a file.

An entity is the data that stores information about the system in a database. Examples of an entity include real world objects, transactions, and persons.

Data attributes

Data attributes describe the properties of a data object. Attributes that identify entities are known as **key attributes**. On the other hand, attributes that describe an entity are known as **non-key attributes**. Generally, a data attribute is used to perform the following functions:

- Naming an instance (occurrence) of data object.
- Description of the instance.
- Making reference to another instance in another table.

Data attributes help to identify and classify an occurrence of entity or a relationship. These attributes represent the information required to develop software and there can be several attributes for a single entity. For example, attributes of 'account' entity are 'number', 'balance', and so on. Similarly, attributes of 'user' entity are 'name', 'address', and 'age'. However, it is important to consider the maximum attributes during requirements elicitation because with more attributes, it is easier for a software development team to develop a software. In case some of the data attributes are not applicable, they can be discarded at a later stage.

Relationships

Entities are linked to each other in different ways. This link or connection of data objects or entities with each other is known as **relationship**. Note that there should be at least two entities to establish a relationship between them. Once the entities are identified, the software development team checks whether relationship exists between them or not. Each relationship has a name, modality or optionality (whether the relationship is optional or mandatory), and degree (number of entities participating in the relationship). These attributes confirm the validity of a given relationship.

To understand *entities*, *data attributes*, and *relationship*, let us consider an example. Suppose in a computerized banking system, one of the processes is to use saving account, which includes two entities, namely, 'user' and 'account'. Each 'user' has a unique 'account number', which makes it easy for the bank to refer to a particular registered user. On the other hand, 'account' entity is used to deposit cash and cheque and to withdraw cash from the saving account. Depending upon the type and nature of transactions, an account can be of various types such as current account, saving account, or over-draft account. The relationship between the user and the account can be described as 'user has account in a bank'.

In Figure 5.2, entities are represented by rectangles, attributes are represented by ellipses, and relationships are represented by diamond symbols. A key attribute is also depicted by an ellipse but with a line below it. This line below the text in the ellipse indicates the uniqueness of each entity.

NOTES

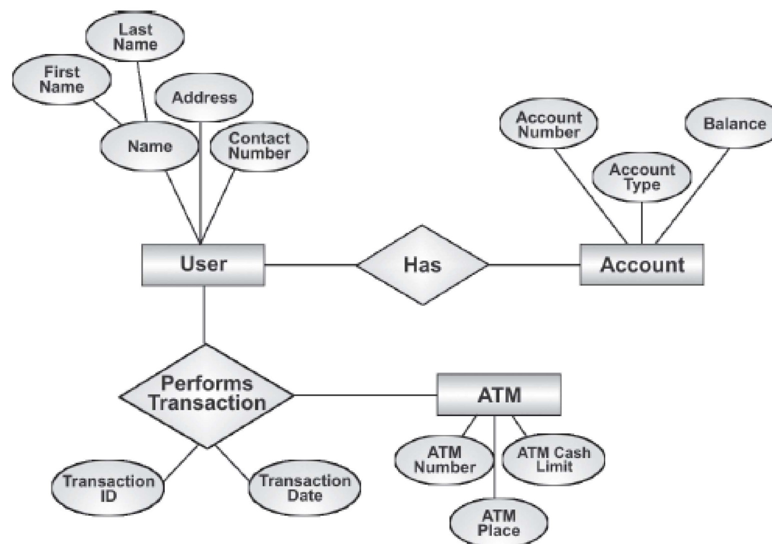


Fig. 5.2 *ER Diagram of Banking System*

Cardinality and Modality

Although data objects, data attributes, and relationships are essential for structured analysis, additional information about them is required to understand the information domain of the problem. This information includes *cardinality* and *modality*. **Cardinality** specifies the number of occurrences (instances) of one data object or entity that relates to the number of occurrence of another data object or entity. There exist three types of cardinality, which are as follows:

- **One-to-one (1:1):** Indicates that one instance of an entity is related only to one instance of another entity. For example, in a database of users in a bank, each user is related to only one account number.
- **One-to-many relationship (1:M):** Indicates that one instance of an entity is related to several instances of another entity. For example, one user can have many accounts in different banks.
- **Many-to-many relationship (M:M):** Indicates that many instances of entities are related to several instances of another entity. For example, many users can have their accounts in many banks.

Modality (also known as **optionality**) describes the possibility whether a relationship between two or more entities and data objects is required. The modality of a relationship is 0 if the relationship is optional. However, the modality is 1 if an occurrence of the relationship is essential.

To understand the concept of cardinality and modality properly, let us consider an example. In Figure 5.3, 'user' entity is related to 'order' entity. Here, cardinality for 'user' entity indicates that the user places an order whereas modality for 'user' entity indicates that it is necessary for a user to place an order. Cardinality for 'order' indicates that a single user can place many orders whereas modality for 'order' entity indicates that a user can arrive without any 'order'.

NOTES

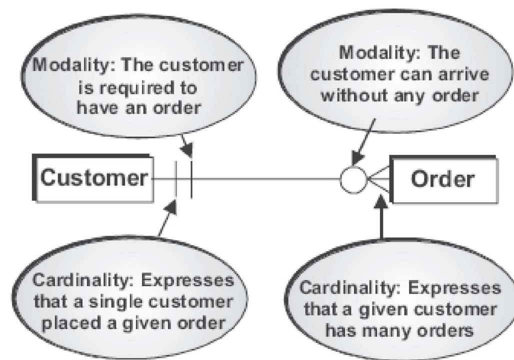


Fig. 5.3 *Cardinality and Modality*

5.4 OBJECT-ORIENTED ANALYSIS

Nowadays, an object-oriented approach is used to describe system requirements using prototypes. This approach is performed using object-oriented modelling (also known as *object-oriented analysis*), which analyses the problem domain and then partitions the problem with the help of objects. An object is an entity that represents a concept and performs a well-defined task in the problem domain. For this, an object contains information of the state and provides services to entities, which are outside the object(s). The state of an object changes when it provides services to other entities.

The object-oriented modelling defines a system as a set of objects, which interact with each other by the services they provide. In addition, objects interact with users through their services so that they can avail the required services in the system. To understand object-oriented analysis, it is important to understand the various concepts used in an object-oriented environment. Some of the commonly used concepts are listed in Table 5.1.

Table 5.1 *Object-Oriented Concepts*

Name	Description
Object	An instance of a class used to describe the entity.
Class	A collection of similar objects, which encapsulate data and procedural abstractions in order to describe their states and the operations to be performed by them.
Attribute	A collection of data values that describe the state of a class.
Operation	Also known as methods and services, it provides a means to modify the state of a class.
Superclass	Also known as base class, it is a generalization of a collection of classes related to it.
Subclass	A specialization of superclass and inherits the attributes and operations from the superclass.
Inheritance	A process in which an object inherits some or all the features of a superclass.
Polymorphism	An ability of objects to be used in more than one form in one or more classes.

NOTES

Generally, it is considered that object-oriented systems are easier to develop and maintain. Also, it is considered that the transition from object-oriented analysis to object-oriented design can be done easily. This is because object-oriented analysis is resilient to changes as objects are more stable than functions that are used in structured analysis. Note that object-oriented analysis comprises a number of steps, which includes identifying objects, identifying structures, identifying attributes, identifying associations and defining services (see Figure 5.4).

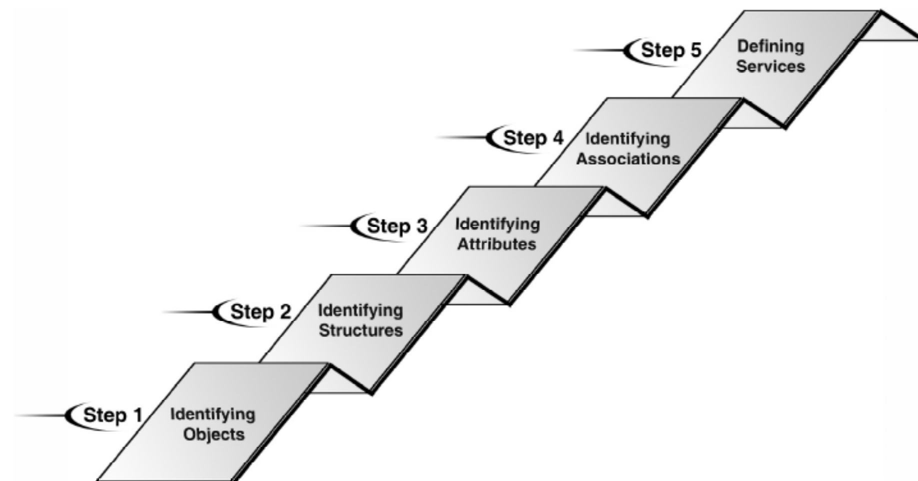


Fig. 5.4 *Steps in Object-Oriented Analysis*

Identifying objects

While performing an analysis, an object encapsulates the attributes on which it provides the services. Note that an object represents entities in a problem domain. The identification of the objects starts by viewing the problem space and its description. Then, a summary of the problem space is gathered to consider the 'nouns'. Nouns indicate the entities used in problem space and which will further be modelled as objects. Some examples of nouns that can be modelled as objects are structures, events, roles and locations.

Identifying structures

Structures depict the hierarchies that exist between the objects. Object modelling applies the concept of generalization and specialization to define hierarchies and to represent the relationships between the objects. As mentioned earlier, superclass is a collection of classes that can further be refined into one or more subclasses. Note that a subclass can have its own attributes and services apart from the attributes and services inherited from its superclass. To understand generalization and specialization, consider an example of class 'car'. Here, 'car' is a superclass, which has attributes, such as wheels, doors and windows. There may be one or more subclasses of a superclass. For instance, a superclass 'car' has subclasses 'Mercedes' and 'Toyota', which have the inherited attributes along with their own attributes, such as comfort, locking system, and so on.

It is essential to consider the objects that can be identified as generalization so that the classification of structure can be identified. In addition, the objects in the problem domain should be determined to check whether they could be classified into specializations or not. Note that the specialization should be meaningful for the problem domain.

Identifying attributes

Attributes add details about an object and store the data for the object. For example, the class 'book' has attributes, such as author name, ISBN and publication house. The data about these attributes is stored in the form of values and are hidden from outside the objects. However, these attributes are accessed and manipulated by the service functions used for that object. The attributes to be considered about an object depend on the problem and the requirement for that attribute. For example, while modelling the student admission system, attributes, such as age and qualification are required for the object 'student'. On the other hand, while modelling for hospital management system, the attribute 'qualification' is unnecessary and requires other attributes of class 'student', such as gender, height and weight. In short, it can be said that while using an object, only the attributes that are relevant and required by the problem domain should be considered.

Identifying associations

Associations describe the relationship between the instances of several classes. For example, an instance of class 'university' is related to an instance of class 'person' by 'educates' relationship. Note that there is no relationship between the class 'university' and class 'person'. However, only the instance(s) of class 'person' (that is, student) is related to class 'university'. This is similar to entity–relationship modelling, where one instance can be related by 1:1, 1:M, and M:M relationships.

An association may have its own attributes, which may or may not be present in other objects. Depending on the requirement, the attributes of the association can be 'forced' to belong to one or more objects without losing the information. However, this should not be done unless the attribute itself belongs to that object.

Defining services

As mentioned earlier, an object performs some services. These services are carried out when the object receives a message for it. Services are a medium to change the state of an object or carry out a process. These services describe the tasks and processes provided by a system. It is important to consider the 'occur' services in order to create, destroy and maintain the instances of an object. To identify the services, the system states are defined and then the external events and the required responses are described. For this, the services provided by objects should be considered.

Check Your Progress

3. Define ER diagram.
4. What is object oriented modeling?

NOTES

5.5 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Requirements elicitation (also known as **requirements capture** and **requirements acquisition**) is a process of collecting information about software requirements from different individuals, such as users and other stakeholders.
2. Prototypes help to clarify unclear requirements. Like scenarios, prototypes also help users to understand the information they need to provide to software development team.
3. IEEE defines ER diagram as '*a diagram that depicts a set of real-world entities and the logical relationships among them'*.
4. The object-oriented modelling defines a system as a set of objects, which interact with each other by the services they provide. In addition, objects interact with users through their services so that they can avail the required services in the system.

5.6 SUMMARY

- Requirements elicitation is a process of collecting information about software requirements from different individuals, such as users and other stakeholders.
- Requirements analysis helps to understand, interpret, classify, and organize the software requirements in order to assess the feasibility, completeness, and consistency of the requirements.
- The data model depicts three interrelated aspects of the system: data objects that are to be processed, attributes that characterize the data objects, and the relationship that links different data objects.
- Entities are linked to each other in different ways. This link or connection of data objects or entities with each other is known as relationship.
- Data objects, data attributes, and relationships are essential for structured analysis, additional information about them is required to understand the information domain of the problem. This information includes cardinality and modality.

NOTES

- An object-oriented approach is used to describe system requirements using prototypes. This approach is performed using object-oriented modelling, which analyses the problem domain and then partitions the problem with the help of objects.
- Structures depict the hierarchies that exist between the objects. Object modelling applies the concept of generalization and specialization to define hierarchies and to represent the relationships between the objects.
- While performing an analysis, an object encapsulates the attributes on which it provides the services.

5.7 KEY WORDS

- **One-to-One (1:1):** Indicates that one instance of an entity is related only to one instance of another entity.
- **One-to-Many Relationship (1:M):** Indicates that one instance of an entity is related to several instances of another entity.
- **Many-to-Many Relationship (M:M):** Indicates that many instances of entities are related to several instances of another entity.

5.8 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What are the some problems that may arise during software elicitation?
2. What do you understand by requirement analysis?
3. What are the elements of requirement model?

Long Answer Questions

1. Explain the various types of elicitation techniques.
2. What are the requirements modeling approaches?
3. Write the significance of data modeling.

5.9 FURTHER READINGS

Pressman, Roger S. 1997. *Software Engineering, a Practitioner's Approach*. New Delhi: Tata McGraw-Hill.

Jalote, Pankaj. 1991. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

Schach, Stephen R. 2005. *Object Oriented and Classical Software Engineering*. New Delhi: Tata McGraw-Hill.

Somerville, Ian. 2001. *Software Engineering*. New Delhi: Pearson Education.

Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandriolli . 1991. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hill of India.

Jawadekar, Waman S. 2004. *Software Engineering: Principles and Practice*. New Delhi: Tata McGraw-Hill.

NOTES

UNIT 6 MODELING

NOTES

Structure

6.0 INTRODUCTION

In this unit, you will learn about the scenario-based modeling, flow modeling, class modeling and behavioral modeling. Software models are ways of expressing a software design. Usually some sort of abstract language or pictures are used to express the software design.

6.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand scenario-based modeling
- Explain flow based modeling
- Explain the class and behavioral modeling

6.2 SCENARIO-BASED MODELING

While developing a software, it is essential for the development team to consider user satisfaction as a top priority to make the software successful. For this, the development team needs to understand how users will interact with the system. This information helps the team to carefully characterize each user requirement and then create a meaningful and relevant analysis model and design model. For this, the software engineer creates scenarios in the form of use-cases to describe the system from users' perspective.

NOTES

Use-case diagrams

Use-cases describe the tasks or series of tasks in which the users will use the software under a specific set of conditions. Each use-case provides one or more scenarios in order to understand how a system should interact with another system to accomplish the required task. Note that use-cases do not provide description about the implementation of software.

Use-cases are represented with the help of a use-case diagram, which depicts the relationships among actors and use-cases within a system. A use-case diagram describes what exists outside the system (actors) and what should be performed by the system (use-cases). The notations used to represent a use-case diagram are listed in Table 6.1.

Table 6.1 *Use-Case Notations*

Name	Symbol	Description
Actor		Relates to the roles people play in an organization or a project. Actors are different kinds of users who use the system in various ways. For example, one actor can be library user whereas another user can be part of library staff.
Use-case		Describes a specific instance of a system function.
Communication link		Indicates the interaction between actor and the system. Communication link is the default line used in a use-case diagram.
Use link	«uses»	Indicates that one of the use-case uses the behaviour described by another use-case.

Let us consider the example of an online registration of a bank to understand how users interact with the system. In Figure 6.1, use-case diagram represents two actors that interact within the system. These actors are user (who interacts with the system to register in the bank) and the bank staff (who provides facilities to the user). There are several use-cases in this diagram, namely, request for registration, display registration form, edit user database, add user name and date, and update.

NOTES

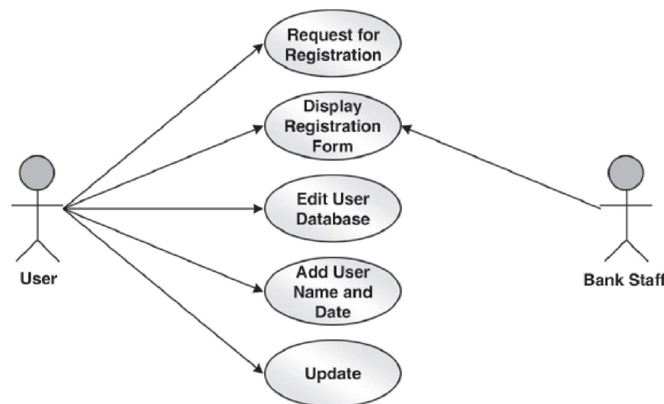


Fig. 6.1 *Use-case Diagram*

6.3 FLOW MODELING

Generally, every system performs three basic functions: *input*, *processing*, and *output*. The system may accept input in various forms such as a file on disk, packet from a network, and so on. The system processes this input by performing some numerical calculations, by applying complex algorithms, and so on to produce the output. The output from a system may also be in different formats such as text file, audio, or video form. The flow model represents the essential functionalities of a system. It depicts the data-flow and data processing along with the control-flow and control processing throughout the system. The flow model is represented by data-flow diagrams and control-flow diagrams. Here, we will discuss only the data-flow diagrams.

Data-flow diagram (DFD)

IEEE defines a data-flow diagram (also known as bubble chart and work flow diagram) as '*a diagram that depicts data sources, data sinks, data storage, and processes performed on data as nodes, and logical flow of data as links between the nodes*'. DFD allows the software development team to depict flow of data from one or more processes to another. In addition, DFD accomplishes the following objectives:

- It represents system data in hierarchical manner and with required level of detail.
- It depicts processes according to defined user requirements and software scope.

A DFD depicts the flow of data within a system and considers a system that transforms inputs into the required outputs. When there is complexity in a system, data needs to be transformed using various steps to produce an output. These steps are required to refine the information. The objective of DFD is to provide an overview of the transformations that occur in the input data within the system in order to produce an output.

NOTES

A DFD should not be confused with a flowchart. A DFD represents the flow of *data* whereas flowchart depicts the flow of *control*. Also, a DFD does not depict the information about the procedure to be used for accomplishing the task. Hence, while making a DFD, procedural details about the processes should not be shown. DFD helps a software designer to describe the transformations taking place in the path of data from input to output.

DFD consists of four basic notations (symbols), which help to depict information in a system. These notations are listed in Table 6.2.

Table 6.2 *DFD Notations*

Name	Notation	Description
External entity	(rectangle)	Represents the source or destination of data within the system. Each external entity is identified with a meaningful and unique name.
Data-flow	(arrows)	Represents the movement of data from its source to destination within the system.
Data store	(two parallel lines)	Indicates the place for storing information within the system.
Process	(circle)	Shows a transformation or manipulation of data within the system. A process is also known as bubble.

While creating a DFD, certain guidelines are followed to depict the data-flow of system requirements effectively. These guidelines help to create DFD in an understandable manner. The commonly followed guidelines for creating DFD are:

- DFD notations should be given meaningful names. For example, verbs should be used for naming a process whereas nouns should be used for naming external entity, data store, and data-flow.
- Abbreviations should be avoided in DFD notations.
- Each process should be numbered uniquely but the numbering should be consistent.
- A DFD should be created in an organized manner so that it is easily understood.
- Unnecessary notations should be avoided in DFD in order to avoid complexity.
- A DFD should be logically consistent. For this, processes without any input or output and any input without output should be avoided.
- There should be no loops in a DFD.
- A DFD should be refined until each process performs a simple function so that it can be easily represented as a program component.
- A DFD should be organized in a series of levels so that each level provides more detail than the previous level.

- The name of a process should be carried to the next level of DFD.
- Each DFD should not have more than six processes and related data stores.
- The data store should be depicted at the context level where it first describes an interface between two or more processes. Then, the data store should be depicted again in the next level of DFD that describes the related processes.

NOTES

There are various levels of DFD, which provide details about the input, processes, and output of a system. Note that the level of detail of process increases with increase in level(s). However, these levels do not describe the system's internal structure or behaviour. These levels are:

- **Level 0 DFD** (also known as context diagram): This shows an overall view of the system.
- **Level 1 DFD**: This elaborates level 0 DFD and splits the process into a detailed form.
- **Level 2 DFD**: This elaborates level 1 DFD and displays the process(s) in a more detailed form.
- **Level 3 DFD**: This elaborates level 2 DFD and displays the process(s) in a detailed form.

To understand various levels of DFD, let us consider an example of banking system. In Figure 6.2, level 0 DFD is drawn. This DFD represents how a 'user' entity interacts with a 'banking system' process and avails its services. The level 0 DFD depicts the entire banking system as a single process. There are various tasks performed in a bank, such as transaction processing, pass book entry, registration, demand draft creation, and online help. The data-flow indicates that these tasks are performed by both the user and the bank. Once the user performs a transaction, the bank verifies whether the user is registered in the bank or not.

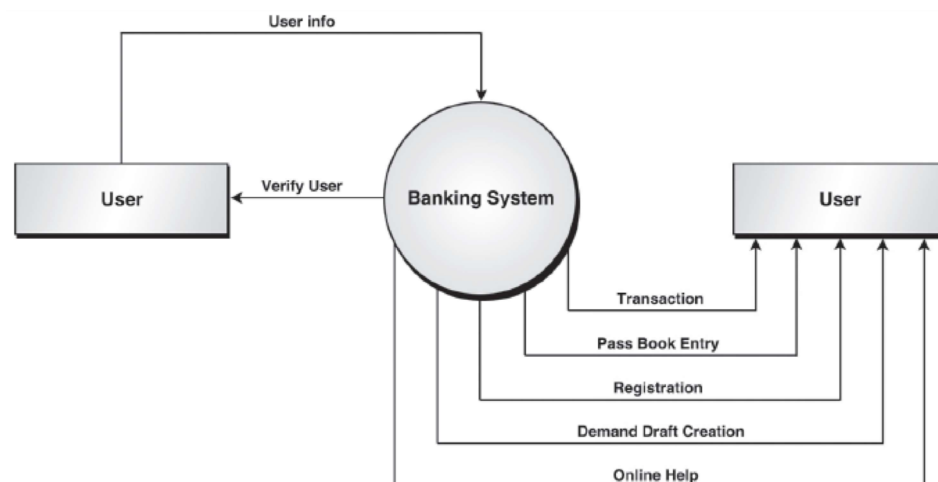


Fig. 6.2 *Level 0 DFD of Banking System*

NOTES

The level 0 DFD is expanded in level 1 DFD (see Figure 6.3). In this DFD, the 'user' entity is related to several processes in the bank, which include 'register', 'user support', and 'provide cash'. Transaction can be performed if the user is already registered in the bank. Once the user is registered, he can perform transaction by the processes, namely, 'deposit cheque', 'deposit cash', and 'withdraw cash'. Note that the line in the process symbol indicates the level of process and contains a unique identifier in the form of a number. If the user is performing transaction to deposit cheque, the user needs to provide a cheque to the bank. The user's information, such as name, address, and account number is stored in 'user_detail' data store, which is a database. If cash is to be deposited and withdrawn, then, the information about the deposited cash is stored in 'cash_detail' data store. User can get a demand draft created by providing cash to the bank. It is not necessary for the user to be registered in that bank to have a demand draft. The details of amount of cash and date are stored in 'DD_detail' data store. Once the demand draft is prepared, its receipt is provided to the user. The 'user support' process helps users by providing answers to their queries related to the services available in the bank.

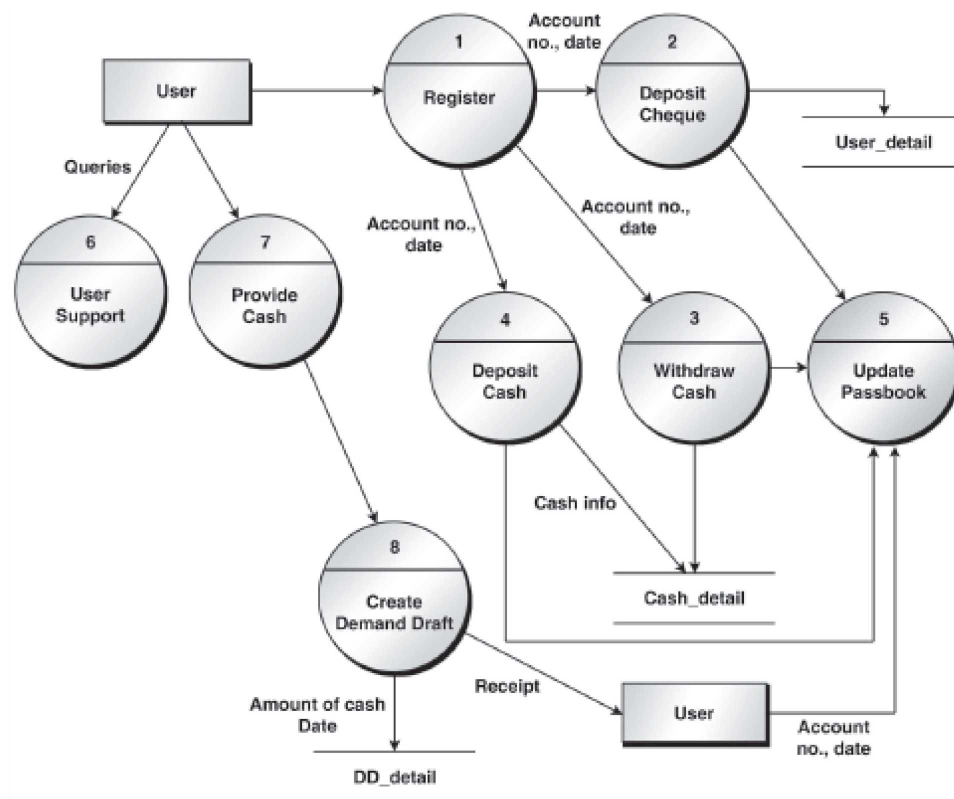


Fig. 6.3 *Level 1 DFD to Perform Transaction*

Level 1 DFD can be further refined into level 2 DFD for any process of banking system that has detailed tasks to perform. For instance, level 2 DFD can be prepared to deposit a cheque, deposit cash, withdraw cash, provide user

support, and to create a demand draft. However, it is important to maintain the continuity of information between the previous levels (level 0 and level 1) and level 2 DFD. As mentioned earlier, the DFD is refined until each process performs a simple function, which is easy to implement.

Let us consider the 'withdraw cash' process (as shown in Figure 6.3) to illustrate the level 2 DFD. The information collected from level 1 DFD acts as an *input* to level 2 DFD. Note that 'withdraw cash' process is numbered as '3' in Figure 6.3 and contains further processes, which are numbered as '3.1', '3.2', '3.3', and '3.4' in Figure 6.4. These numbers represent the sublevels of 'withdraw cash' process. To withdraw cash, the bank checks the status of balance in the user's account (as shown by 'check account status' process) and then allots a token (shown as 'allot token' process). After the user withdraws cash, the balance in user's account is updated in the 'user_detail' data store and a statement is provided to the user.

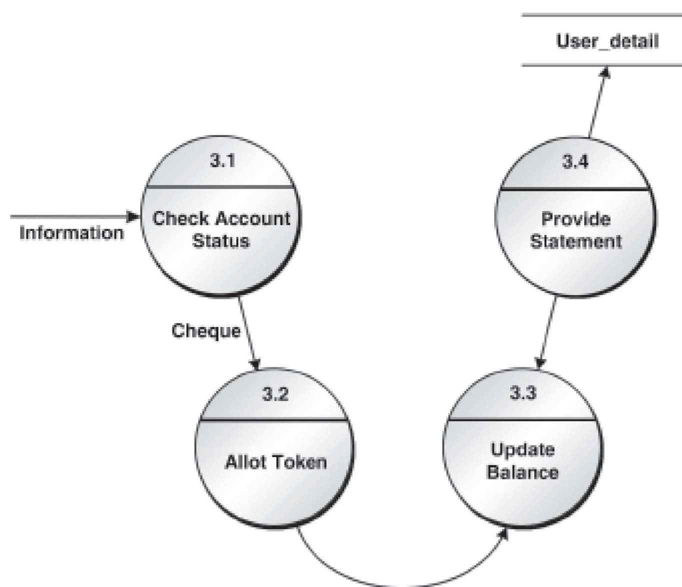


Fig. 6.4 *Level 2 DFD to Withdraw Cash*

If a particular process of level 2 DFD requires elaboration, then this level is further refined into level 3 DFD. Let us consider the process 'check account status' (see Figure 6.4) to illustrate level 3 DFD. In Figure 6.5, this process contains further processes numbered as '3.1.1' and '3.1.2', which describe the sublevels of 'check account status' process. To check the account status, the bank fetches the account detail (shown as 'fetch account detail' process) from the 'account_detail' data store. After fetching the details, the balance is read (shown as 'read balance' process) from the user's account. Note that the requirements engineering process of DFDs continues until each process performs a function that can be easily implemented as an individual program component.

NOTES

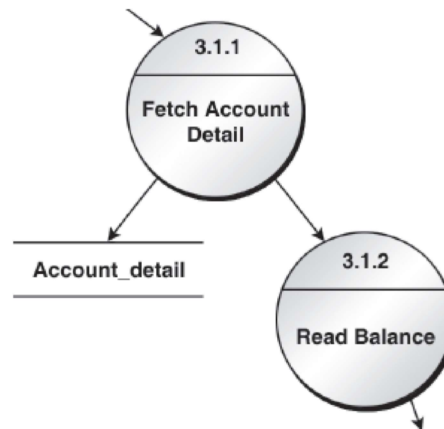


Fig. 6.5 *Level 3 DFD to Check Account Status*

Data dictionary

Although data-flow diagrams contain meaningful names of notations, they do not provide complete information about the structure of data-flows. For this, a data dictionary is used, which is a *repository* that stores description of data objects to be used by the software. A data dictionary stores an organized collection of information about data and their relationships, data-flows, data types, data stores, processes, and so on. In addition, it helps users to understand the data types and processes defined along with their uses. It also facilitates the validation of data by avoiding duplication of entries and provides online access to definitions to the users.

A data dictionary comprises the source of data, which are data objects and entities as well as the elements listed here:

- **Name**: Provides information about the primary name of the data store, external entity, and data-flow.
- **Alias**: Describes different names of data objects and entities used.
- **Where-used/how-used**: Lists all the processes that use data objects and entities and how they are used in the system. For this, it describes the inputs to the process, output from the process, and the data store.
- **Content description**: Provides information about the content with the help of data dictionary notations (such as '=', '+', and '* *').
- **Supplementary information**: Provides information about data types, values used in variables, and limitations of these values.

6.4 CLASS MODELING

Once all the objects in the problem domain are identified, the objects that have the same characteristics are grouped together to form an object class or simply a

class. A **class** is a type definition that defines what attributes each object of that class encapsulate and what services it provides. Note that class is just a definition and it does not create any object and cannot hold any value. When objects of a class are created, memory for them is allocated.

NOTES

During class modeling, all the classes are identified and represented using UML class diagrams. The notation for a class is a rectangular box divided into three sections. The top section contains the class name, middle section contains the attributes that belong to the class, and the bottom section lists all the services (operations) provided by the objects of this class. The UML convention is to use boldface for class name and to capitalize the initial letter of each word in the class name (see Figure 6.6).

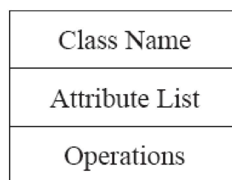


Fig. 6.6 *Class Notation in UML*

Figure 6.7 shows an example of class diagram depicting *Student* class, which has the attributes *Roll no*, *Name*, *Marks* and *Grade*. The operations that can be performed on these attributes are *Input_Details()*, *Output_Details()*, and *Compute_Grade()*.

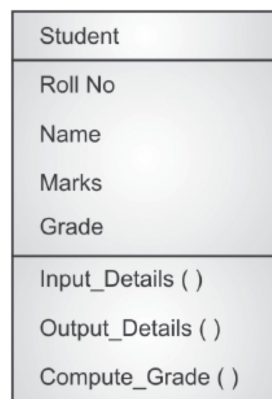


Fig. 6.7 *Class Diagram for Student Class*

The inheritance relationship between classes is modeled by using generalization-specialization structure. Generalization defines a hierarchy of abstraction in which one or more subclasses are inherited from one or more super classes. In generalization structure, a subclass inherits all or some attributes and services of a super (or general) class. In addition, the subclass can add more attributes and services specific to that class. Thus, by using a general class and inheritance, one can create a specialized version of the general class. It is also called "is-a-kind-of" relationship. For example, *Graduate Student* is a kind of

NOTES

Student. Generalization can be shown in a class diagram by a triangle connecting a super class to its subclasses (see Figure 6.8).

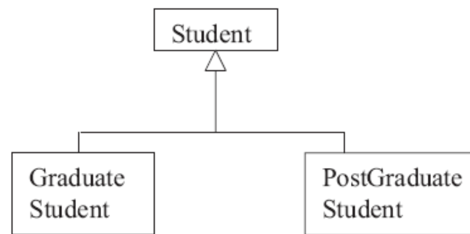


Fig. 6.8 *Generalization of Student Class*

Specialization is just the reverse of generalization in a sense that if a subclass is seen from a super class, the subclass is seen as a specialized version of super class, and in reverse, a super class looks like a generalized version of subclass. For example, in the above hierarchy, the classes *Graduate Student* and *PostGraduate Student* are the specialization of the class *Student* or that the class *Student* is generalization of classes *Graduate Student* and *PostGraduate Student*.

In addition to generalization-specialization, classes also have one more basic relationship, which is aggregation. Aggregation refers to a component class and it models the "part-whole" or "is-a-part-of" relationship between a class and its components. For example, with the class *Student*, *Exam Schedule* class as a component may exist. This relationship between them can be represented by using a diamond to denote aggregation as shown in Figure 6.9.

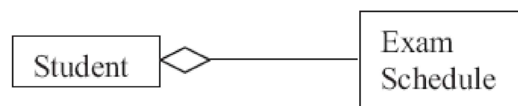


Fig. 6.9 *Representation of Aggregation Relationship*

6.5 BEHAVIORAL MODELING

Behavioral model depict the overall behavior of the system. A system always exists in some **state**—an observable mode of behavior; and it may change its state in response to some event occurrence in the external environment. The behavioral model is represented using a state transition diagram.

State transition diagram

The state transition diagram shows the system behavior by depicting its states and the events that cause the change in system state. To better understand the design of state transition diagram, consider an object that represents a queue. Figure 6.10 shows the state transition diagram of this queue object. Initially, the queue is empty and a delete operation on an empty queue results in an error. However,

when the first element is inserted into the queue, it reaches to intermediate state, which is not-empty-not-full. Further insertions will keep the queue in this state until the maximum size is reached. At this point, the state of the queue changes and it becomes full. An insertion on the full queue results in an error. The delete operation can be performed on full as well as not-empty-not-full queue. When an element is deleted from the full queue, the queue becomes not-empty-not-full. Further deletions keep the queue in this state, however, when the last element is deleted from the queue, the queue becomes empty.

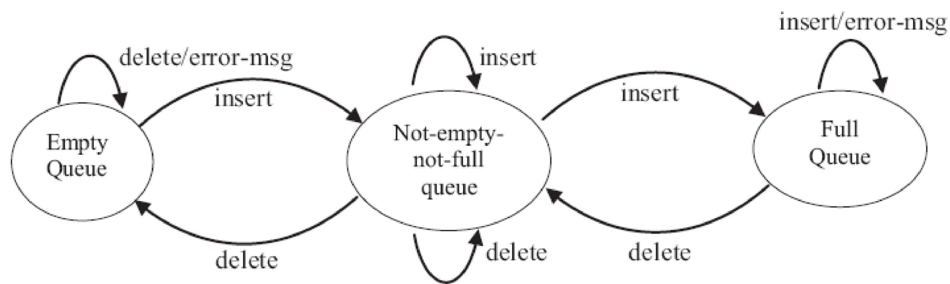


***Fig. 6.10** State Transition Diagram for Queue Object*

Check Your Progress

1. Define Data flow diagram (DFD).
2. What is Data dictionary?
3. Define class modelling.

6.6 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. DFD is a diagram that depicts data sources, data sinks, data storage, and processes performed on data as nodes, and logical flow of data as links between the nodes.
2. A data dictionary stores an organized collection of information about data and their relationships, data-flows, data types, data stores, processes.
3. A class is a type definition that defines what attributes each object of that class encapsulate and what services it provides.

6.7 SUMMARY

- Use-cases describe the tasks or series of tasks in which the users will use the software under a specific set of conditions.
- Use-cases are represented with the help of a use-case diagram, which depicts the relationships among actors and use-cases within a system. A use-case

NOTES

diagram describes what exists outside the system (actors) and what should be performed by the system (use-cases).

- A DFD depicts the flow of data within a system and considers a system that transforms inputs into the required outputs. When there is complexity in a system, data needs to be transformed using various steps to produce an output.
- DFD is a diagram that depicts data sources, data sinks, data storage, and processes performed on data as nodes, and logical flow of data as links between the nodes.
- A data dictionary stores an organized collection of information about data and their relationships, data-flows, data types, data stores, processes.
- A class is a type definition that defines what attributes each object of that class encapsulate and what services it provides. Note that class is just a definition and it does not create any object and cannot hold any value.
- Behavioral model depict the overall behavior of the system. A system always exists in some state—an observable mode of behavior; and it may change its state in response to some event occurrence in the external environment.
- The state transition diagram shows the system behavior by depicting its states and the events that cause the change in system state. To better understand the design of state transition diagram, consider an object that represents a queue.

6.8 KEY WORDS

- **Data Flow Diagram:** It is a way of representing a flow of a data of a process or a system.
- **Class:** It is a type definition that defines what attributes each object of that class encapsulate and what services it provides.
- **Data dictionary:** A repository that stores description of data objects to be used by the software.

6.9 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What do you understand by use-case diagram?
2. Write a note on DFD.
3. Discuss the notations used in DFD.

Long Answer Questions

1. Explain the various levels of DFD with the help of examples.
2. Explain the class based modeling.
3. Write a detailed note on behavioral modeling.

NOTES

6.10 FURTHER READINGS

Pressman, Roger S. 1997. *Software Engineering, a Practitioner's Approach*. New Delhi: Tata McGraw-Hill.

Jalote, Pankaj. 1991. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

Schach, Stephen R. 2005. *Object Oriented and Classical Software Engineering*. New Delhi: Tata McGraw-Hill.

Somerville, Ian. 2001. *Software Engineering*. New Delhi: Pearson Education.

Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandriolli . 1991. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hill of India.

Jawadekar, Waman S. 2004. *Software Engineering: Principles and Practice*. New Delhi: Tata McGraw-Hill.

BLOCK - III
DESIGN

NOTES

UNIT 7 DESIGN ENGINEERING

Structure

7.0 INTRODUCTION

Once the requirements document for the software to be developed is available, the software design phase begins. While the requirement specification activity deals entirely with the problem domain, design is the first phase of transforming the problem into a solution. In the design phase, the customer and business requirements and technical considerations all come together to formulate a product or a system.

The design process comprises a set of principles, concepts and practices, which allow a software engineer to model the system or product that is to be built. This model, known as design model, is assessed for quality and reviewed before a code is generated and tests are conducted. The design model provides details about software data structures, architecture, interfaces and components which are required to implement the system.

7.1 OBJECTIVES

After going through this unit, you will be able to:

- Discuss the principles of software design
- Understand the concept of software design
- Explain the different types of design patterns
- Develop a design model

NOTES

7.2 BASICS OF SOFTWARE DESIGN

Software design is a phase in software engineering, in which a blueprint is developed to serve as a base for constructing the software system. **IEEE** defines software design as '*both a process of defining the architecture, components, interfaces, and other characteristics of a system or component and the result of that process*.'

In the design phase, many critical and strategic decisions are made to achieve the desired functionality and quality of the system. These decisions are taken into account to successfully develop the software and carry out its maintenance in a way that the quality of the end product is improved.

Principles of Software Design

Developing design is a cumbersome process as most expansive errors are often introduced in this phase. Moreover, if these errors get unnoticed till later phases, it becomes more difficult to correct them. Therefore, a number of principles are followed while designing the software. These principles act as a framework for the designers to follow a good design practice (see Figure 7.1).

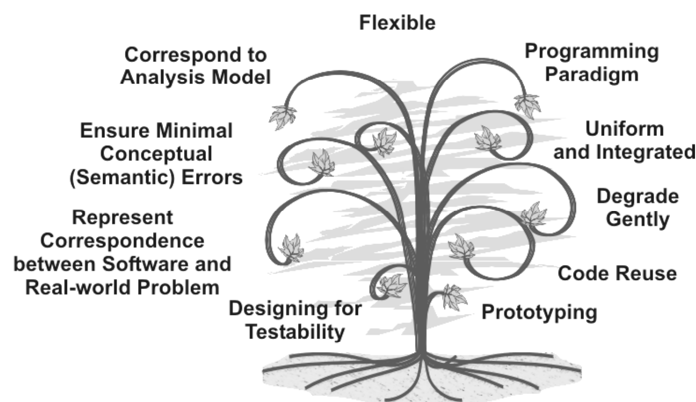


Fig. 7.1 *Principles of Software Design*

Some of the commonly followed design principles are as following.

- **Software design should correspond to the analysis model:** Often a design element corresponds to many requirements, therefore, we must know how the design model satisfies all the requirements represented by the analysis model.
- **Choose the right programming paradigm:** A programming paradigm describes the structure of the software system. Depending on the nature and type of application, different programming paradigms such as procedure oriented, object-oriented, and prototyping paradigms can be used. The paradigm should be chosen keeping constraints in mind such as time, availability of resources and nature of user's requirements.

NOTES

- **Software design should be uniform and integrated:** Software design is considered uniform and integrated, if the interfaces are properly defined among the design components. For this, rules, format, and styles are established before the design team starts designing the software.
- **Software design should be flexible:** Software design should be flexible enough to adapt changes easily. To achieve the flexibility, the basic design concepts such as abstraction, refinement, and modularity should be applied effectively.
- **Software design should ensure minimal conceptual (semantic) errors:** The design team must ensure that major conceptual errors of design such as ambiguousness and inconsistency are addressed in advance before dealing with the syntactical errors present in the design model.
- **Software design should be structured to degrade gently:** Software should be designed to handle unusual changes and circumstances, and if the need arises for termination, it must do so in a proper manner so that functionality of the software is not affected.
- **Software design should represent correspondence between the software and real-world problem:** The software design should be structured in such a way that it always relates with the real-world problem.
- **Software reuse:** Software engineers believe on the phrase: '*do not reinvent the wheel*'. Therefore, software components should be designed in such a way that they can be effectively reused to increase the productivity.
- **Designing for testability:** A common practice that has been followed is to keep the testing phase separate from the design and implementation phases. That is, first the software is developed (designed and implemented) and then handed over to the testers who subsequently determine whether the software is fit for distribution and subsequent use by the customer. However, it has become apparent that the process of separating testing is seriously flawed, as if any type of design or implementation errors are found after implementation, then the entire or a substantial part of the software requires to be redone. Thus, the test engineers should be involved from the initial stages. For example, they should be involved with analysts to prepare tests for determining whether the user requirements are being met.
- **Prototyping:** Prototyping should be used when the requirements are not completely defined in the beginning. The user interacts with the developer to expand and refine the requirements as the development proceeds. Using prototyping, a quick 'mock-up' of the system can be developed. This mock-up can be used as a effective means to give the users a feel of what the system will look like and demonstrate functions that will be included in the developed system. Prototyping also helps in reducing risks of designing software that is not in accordance with the customer's requirements.

NOTES

Note that design principles are often constrained by the existing hardware configuration, the implementation language, the existing file and data structures, and the existing organizational practices. Also, the evolution of each software design should be meticulously designed for future evaluations, references and maintenance.

7.2.1 Software Design Concepts

Every software process is characterized by basic concepts along with certain practices or methods. Methods represent the manner through which the concepts are applied. As new technology replaces older technology, many changes occur in the methods that are used to apply the concepts for the development of software. However, the fundamental concepts underlining the software design process remain the same, some of which are described here.

Abstraction

Abstraction refers to a powerful design tool, which allows software designers to consider components at an abstract level, while neglecting the implementation details of the components. **IEEE** defines abstraction as '*a view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information.*' The concept of abstraction can be used in two ways: as a process and as an entity. As a **process**, it refers to a mechanism of hiding irrelevant details and representing only the essential features of an item so that one can focus on important things at a time. As an **entity**, it refers to a model or view of an item.

Each step in the software process is accomplished through various levels of abstraction. At the highest level, an outline of the solution to the problem is presented whereas at the lower levels, the solution to the problem is presented in detail. For example, in the requirements analysis phase, a solution to the problem is presented using the language of problem environment and as we proceed through the software process, the abstraction level reduces and at the lowest level, source code of the software is produced.

There are three commonly used abstraction mechanisms in software design, namely, *functional abstraction*, *data abstraction* and *control abstraction*. All these mechanisms allow us to control the complexity of the design process by proceeding from the abstract design model to concrete design model in a systematic manner.

- **Functional abstraction:** This involves the use of parameterized subprograms. Functional abstraction can be generalized as collections of subprograms referred to as 'groups'. Within these groups there exist routines which may be visible or hidden. Visible routines can be used within the containing groups as well as within other groups, whereas hidden routines are hidden from other groups and can be used within the containing group only.

NOTES

- **Data abstraction:** This involves specifying data that describes a data object. For example, the data object *window* encompasses a set of attributes (window type, window dimension) that describe the *window* object clearly. In this abstraction mechanism, representation and manipulation details are ignored.
- **Control abstraction:** This states the desired effect, without stating the exact mechanism of control. For example, if and while statements in programming languages (like C and C++) are abstractions of machine code implementations, which involve conditional instructions. In the architectural design level, this abstraction mechanism permits specifications of sequential subprogram and exception handlers without the concern for exact details of implementation.

Architecture

Software architecture refers to the structure of the system, which is composed of various components of a program/system, the attributes (properties) of those components and the relationship amongst them. The software architecture enables the software engineers to analyze the software design efficiently. In addition, it also helps them in decision-making and handling risks. The software architecture does the following.

- Provides an insight to all the interested stakeholders that enable them to communicate with each other
- Highlights early design decisions, which have great impact on the software engineering activities (like coding and testing) that follow the design phase
- Creates intellectual models of how the system is organized into components and how these components interact with each other.

Currently, software architecture is represented in an informal and unplanned manner. Though the architectural concepts are often represented in the infrastructure (for supporting particular architectural styles) and the initial stages of a system configuration, the lack of an explicit independent characterization of architecture restricts the advantages of this design concept in the present scenario. Note that software architecture comprises two elements of design model, namely, *data design* and *architectural design*. Both these elements have been discussed later in this unit.

Patterns

A pattern provides a description of the solution to a recurring design problem of some specific domain in such a way that the solution can be used again and again. The objective of each pattern is to provide an insight to a designer who can determine the following.

- Whether the pattern can be reused

- Whether the pattern is applicable to the current project
- Whether the pattern can be used to develop a similar but functionally or structurally different design pattern.

7.2.2 Types of Design Patterns

Software engineer can use the design pattern during the entire software design process. When the analysis model is developed, the designer can examine the problem description at different levels of abstraction to determine whether it complies with one or more of the following types of design patterns.

- **Architectural patterns:** These patterns are high-level strategies that refer to the overall structure and organization of a software system. That is, they define the elements of a software system such as subsystems, components, classes, etc. In addition, they also indicate the relationship between the elements along with the rules and guidelines for specifying these relationships. Note that architectural patterns are often considered equivalent to *software architecture*.
- **Design patterns**: These patterns are medium-level strategies that are used to solve design problems. They provide a means for the refinement of the elements (as defined by architectural pattern) of a software system or the relationship among them. Specific design elements such as relationship among components or mechanisms that affect component-to-component interaction are addressed by design patterns. Note that design patterns are often considered equivalent to *software components*.
- **Idioms**: These patterns are low-level patterns, which are programming-language specific. They describe the implementation of a software component, the method used for interaction among software components, etc., in a specific programming language. Note that idioms are often termed as coding patterns.

Modularity

Modularity is achieved by dividing the software into uniquely named and addressable *components*, which are also known as **modules**. A complex system (large program) is partitioned into a set of discrete modules in such a way that each module can be developed independent of other modules (see Figure 7.2). After developing the modules, they are integrated together to meet the software requirements. Note that larger the number of modules a system is divided into, greater will be the effort required to integrate the modules.

NOTES

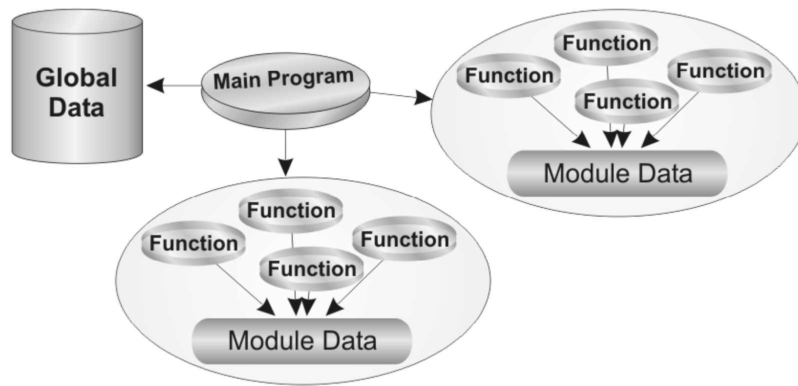


Fig. 7.2 *Modules in Software Programs*

Modularizing a design helps to plan the development in a more effective manner, accommodate changes easily, conduct testing and debugging effectively and efficiently, and conduct maintenance work without adversely affecting the functioning of the software.

Information Hiding

Modules should be specified and designed in such a way that the data structures and processing details of one module are not accessible to other modules (see Figure 7.3). They pass only that much information to each other, which is required to accomplish the software functions. The way of hiding unnecessary details is referred to as **information hiding**. **IEEE** defines information hiding as '*the technique of encapsulating software design decisions in modules in such a way that the module's interfaces reveal as little as possible about the module's inner workings; thus each module is a 'black box' to the other modules in the system*.'

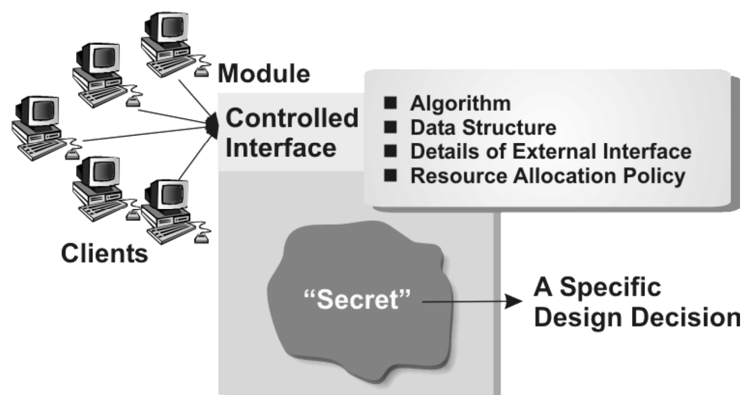


Fig. 7.3 *Information Hiding*

Information hiding is of immense use when modifications are required during the testing and maintenance phase. Some of the advantages associated with information hiding are listed below.

- Leads to low coupling
- Emphasizes communication through controlled interfaces
- Decreases the probability of adverse effects
- Restricts the effects of changes in one component on others
- Results in higher quality software.

Stepwise Refinement

Stepwise refinement is a top-down design strategy used for decomposing a system from a high level of abstraction into a more detailed level (lower level) of abstraction. At the highest level of abstraction, function or information is defined conceptually without providing any information about the internal workings of the function or internal structure of the data. As we proceed towards the lower levels of abstraction, more and more details are available.

Software designers start the stepwise refinement process by creating a sequence of compositions for the system being designed. Each composition is more detailed than the previous one and contains more components and interactions. The earlier compositions represent the significant interactions within the system, while the later compositions show in detail how these interactions are achieved.

To have a clear understanding of the concept, let us consider an example of stepwise refinement. Every computer program comprises *input*, *process,* and *output*.

1. INPUT
   - Get user's name (string) through a prompt.
   - Get user's grade (integer from 0 to 100) through a prompt and validate.
2. PROCESS
3. OUTPUT

This is the first step in refinement. The input phase can be refined further as given here.

1. INPUT
   - Get user's name through a prompt.
   - Get user's grade through a prompt.
   - While (invalid grade)

     Ask again:
2. PROCESS
3. OUTPUT

Note: Stepwise refinement can also be performed for PROCESS and OUTPUT phase.

NOTES

Refactoring

Refactoring is an important design activity that reduces the complexity of module design keeping its behaviour or function unchanged. Refactoring can be defined as a process of modifying a software system to improve the internal structure of design without changing its external behavior. During the refactoring process, the existing design is checked for any type of flaws like redundancy, poorly constructed algorithms and data structures, etc., in order to improve the design. For example, a design model might yield a component which exhibits low cohesion (like a component performs four functions that have a limited relationship with one another). Software designers may decide to refactor the component into four different components, each exhibiting high cohesion. This leads to easier integration, testing, and maintenance of the software components.

Structural Partitioning

When the architectural style of a design follows a hierarchical nature, the structure of the program can be partitioned either horizontally or vertically. In **horizontal partitioning**, the control modules (as depicted in the shaded boxes in Figure 7.4(a)) are used to communicate between functions and execute the functions. Structural partitioning provides the following benefits.

- The testing and maintenance of software becomes easier.
- The negative impacts spread slowly.
- The software can be extended easily.

Besides these advantages, horizontal partitioning has some disadvantage also. It requires to pass more data across the module interface, which makes the control-flow of the problem more complex. This usually happens in cases where data moves rapidly from one function to another.

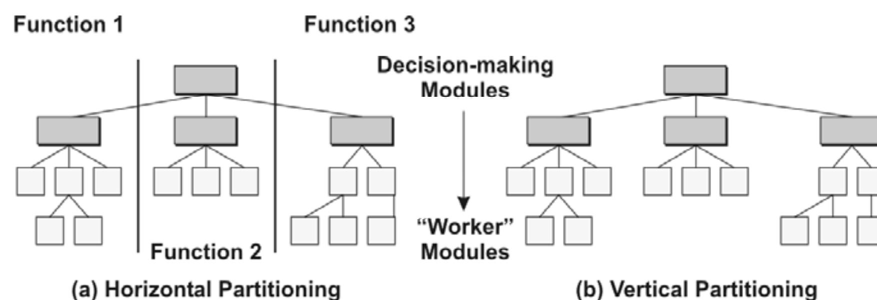


Fig. 7.4 *Horizontal and Vertical Partitioning*

In **vertical partitioning** (see Figure 7.4 (b)), the functionality is distributed among the modules in a top-down manner. The modules at the top level called **control modules** perform the decision-making and do little processing whereas the modules at the low level called **worker modules** perform all input, computation and output tasks.

NOTES

Concurrency

Computer has limited resources and they must be utilized efficiently as much as possible. To utilize these resources efficiently, multiple tasks must be executed concurrently. This requirement makes concurrency one of the major concepts of software design. Every system must be designed to allow multiple processes to execute concurrently, whenever possible. For example, if the current process is waiting for some event to occur, the system must execute some other process in the mean time.

However, concurrent execution of multiple processes sometimes may result in undesirable situations such as an inconsistent state, deadlock, etc. For example, consider two processes A and B and a data item Q1 with the value '200'. Further, suppose A and B are being executed concurrently and firstly A reads the value of Q1 (which is '200') to add '100' to it. However, before A updates the value of Q1, B reads the value of Q1 (which is still '200') to add '50' to it. In this situation, whether A or B first updates the value of Q1, the value of Q1 would definitely be wrong resulting in an inconsistent state of the system. This is because the actions of A and B are not synchronized with each other. Thus, the system must control the concurrent execution and synchronize the actions of concurrent processes.

One way to achieve synchronization is mutual exclusion, which ensures that two concurrent processes do not interfere with the actions of each other. To ensure this, mutual exclusion may use locking technique. In this technique, the processes need to lock the data item to be read or updated. The data item locked by some process cannot be accessed by other processes until it is unlocked. It implies that the process, that needs to access the data item locked by some other process, has to wait.

7.2.3 Developing a Design Model

To develop a complete specification of design (design model), four design models are needed (see Figure 7.5). These models are listed below.

- **Data design:** This specifies the data structures for implementing the software by converting data objects and their relationships identified during the analysis phase. Various studies suggest that design engineering should begin with data design, since this design lays the foundation for all other design models.
- **Architectural design:** This specifies the relationship between the structural elements of the software, design patterns, architectural styles, and the factors affecting the ways in which architecture can be implemented.
- **Component-level design:** This provides the detailed description of how structural elements of software will actually be implemented.
- **Interface design:** This depicts how the software communicates with the system that interoperates with it and with the end-users.

NOTES

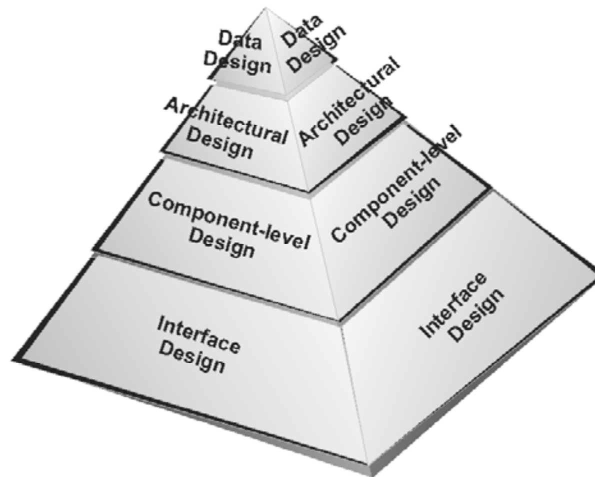


Fig. 7.5 *Design Model and its Elements*

Check Your Progress

1. What is software design?
2. Define abstraction.
3. Discuss modular programming.
4. Define modularity.

7.3 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. IEEE defines software design as '*both a process of defining the architecture, components, interfaces, and other characteristics of a system or component and the result of that process.*'
2. IEEE defines abstraction as '*a view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information.*'
3. Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules.
4. Modularity is achieved by dividing the software into uniquely named and addressable *components*, which are also known as modules. A complex system (large program) is partitioned into a set of discrete modules in such a way that each module can be developed independent of other modules.

7.4 SUMMARY

NOTES

- Software design is a phase in software engineering, in which a blueprint is developed to serve as a base for constructing the software system.
- A process of defining the architecture, components, interfaces, and other characteristics of a system or component and the result of that process.
- Abstraction refers to a powerful design tool, which allows software designers to consider components at an abstract level, while neglecting the implementation details of the components.
- Software architecture refers to the structure of the system, which is composed of various components of a program/system, the attributes (properties) of those components and the relationship amongst them.
- A pattern provides a description of the solution to a recurring design problem of some specific domain in such a way that the solution can be used again and again.
- Modularity is achieved by dividing the software into uniquely named and addressable components, which are also known as modules.
- Software architecture comprises two elements of design model, namely, data design and architectural design.

7.5 KEY WORDS

- **Abstraction**: It is a mechanism to hide irrelevant details and represent only the essential features so that one can focus on important things at a time.
- **Functional Abstraction:** It involves the use of parameterized subprograms. Functional abstraction can be generalized as collections of subprograms referred to as groups.
- **Data Abstraction:** It involves specifying data that describes a data object. The data object window encompasses a set of attributes that describe the window object clearly.
- **Control Abstraction:** This states the desired effect, without stating the exact mechanism of control.

7.6 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Define software design.

2. What do you understand by abstraction?
3. What are the benefits of structural partitioning?

Long Answer Questions

NOTES

1. Explain the various principles of software design.
2. What are the different types of design pattern?
3. Explain how will you develop a design model.

7.7 FURTHER READINGS

Pressman, Roger S. 1997. *Software Engineering, a Practitioner's Approach*. New Delhi: Tata McGraw-Hill.

Jalote, Pankaj. 1991. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

Schach, Stephen R. 2005. *Object Oriented and Classical Software Engineering*. New Delhi: Tata McGraw-Hill.

Somerville, Ian. 2001. *Software Engineering*. New Delhi: Pearson Education.

Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandriolli . 1991. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hill of India.

Jawadekar, Waman S. 2004. *Software Engineering: Principles and Practice*. New Delhi: Tata McGraw-Hill.

UNIT 8 ARCHITECTURAL DESIGN

NOTES

Structure

8.0 INTRODUCTION

In this unit, you will learn about the architectural design of software. Software architecture refers to the structure of the system, which is composed of various components of a program/system, the attributes (properties) of those components and the relationship amongst them. The software architecture enables the software engineers to analyze the software design efficiently.

8.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand the concept of data design
- Define architectural design
- Understand the models for architectural design representation
- Discuss the architectural styles

8.2 DATA DESIGN

Data design is the first design activity, which results in less complex, modular, and efficient program structure. The information domain model developed during analysis phase is transformed into data structures needed for implementing the software. The data objects, attributes, and relationships depicted in entity relationship diagrams and the information stored in data dictionary provide a base for data design activity. During the data design process, data types are specified along with the integrity rules required for the data. For specifying and designing efficient data structures, some principles should be followed. These principles are listed below.

NOTES

- The data structures needed for implementing the software as well as the operations that can be applied on them should be identified.
- A data dictionary should be developed to depict how different data objects interact with each other and what constraints are to be imposed on the elements of data structure.
- Stepwise refinement should be used in data design process and detailed design decisions should be made later in the process.
- Only those modules that need to access data stored in a data structure directly should be aware of the representation of the data structure.
- A library containing the set of useful data structures along with the operations that can be performed on them should be maintained.
- Language used for developing the system should support abstract data types.

The structure of data can be viewed at three levels, namely, *program component level*, *application level*, and *business level*. At the **program component level**, the design of data structures and the algorithms required to manipulate them is necessary, if high-quality software is desired. At the **application level**, it is crucial to convert the data model into a database so that the specific business objectives of a system could be achieved. At the **business level**, the collection of information stored in different databases should be reorganized into data warehouse, which enables data mining that has an influential impact on the business.

Note: Data design helps to represent the data component in the conventional systems and class definitions in object-oriented systems.

8.2.1 Archiectural Design

Requirements of the software should be transformed into an architecture that describes the software's top-level structure and identifies its components. This is accomplished through architectural design (also called **system design**), which acts as a preliminary 'blueprint' from which software can be developed. **IEEE** defines architectural design as '*the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.*' This framework is established by examining the software requirements document and designing a model for providing implementation details. These details are used to specify the components of the system along with their inputs, outputs, functions, and the interaction between them. An architectural design performs the following functions.

- It defines an abstraction level at which the designers can specify the functional and performance behaviour of the system.
- It acts as a guideline for enhancing the system (whenever required) by describing those features of the system that can be modified easily without affecting the system integrity.

- It evaluates all top-level designs.
- It develops and documents top-level design for the external and internal interfaces.
- It develops preliminary versions of user documentation.
- It defines and documents preliminary test requirements and the schedule for software integration.

The sources of architectural design are listed below.

- Information regarding the application domain for the software to be developed
- Using data-flow diagrams
- Availability of architectural patterns and architectural styles.

Architectural design is of crucial importance in software engineering during which the essential requirements like reliability, cost, and performance are dealt with. This task is cumbersome as the software engineering paradigm is shifting from monolithic, stand-alone, built-from-scratch systems to componentized, evolvable, standards-based, and product line-oriented systems. Also, a key challenge for designers is to know precisely how to proceed from requirements to architectural design. To avoid these problems, designers adopt strategies such as reusability, componentization, platform-based, standards-based, and so on.

Though the architectural design is the responsibility of developers, some other people like user representatives, systems engineers, hardware engineers, and operations personnel are also involved. All these stakeholders must also be consulted while reviewing the architectural design in order to minimize the risks and errors.

Architectural Design Representation

Architectural design can be represented using the following models.

- **Structural model:** Illustrates architecture as an ordered collection of program components
- **Dynamic model:** Specifies the behavioral aspect of the software architecture and indicates how the structure or system configuration changes as the function changes due to change in the external environment
- **Process model:** Focuses on the design of the business or technical process, which must be implemented in the system
- **Functional model:** Represents the functional hierarchy of a system
- **Framework model:** Attempts to identify repeatable architectural design patterns encountered in similar types of application. This leads to an increase in the level of abstraction.

NOTES

Architectural Design Output

The architectural design process results in an **Architectural Design Document** (**ADD**). This document consists of a number of graphical representations that comprises software models along with associated descriptive text. The software models include *static model*, *interface model*, *relationship model*, and *dynamic process model*. They show how the system is organized into a process at run-time.

Architectural design document gives the developers a solution to the problem stated in the Software Requirements Specification (SRS). Note that it considers only those requirements in detail that affect the program structure. In addition to ADD, other outputs of the architectural design are listed below.

- Various reports including audit report, progress report, and configuration status accounts report
- Various plans for detailed design phase, which include the following
  - o Software verification and validation plan
  - o Software configuration management plan
  - o Software quality assurance plan
  - o Software project management plan.

8.2.2 Architectural Styles

Architectural styles define a group of interlinked systems that share structural and semantic properties. In short, the objective of using architectural styles is to establish a structure for all the components present in a system. If an existing architecture is to be re-engineered, then imposition of an architectural style results in fundamental changes in the structure of the system. This change also includes re-assignment of the functionality performed by the components.

By applying certain constraints on the design space, we can make different style-specific analysis from an architectural style. In addition, if conventional structures are used for an architectural style, the other stakeholders can easily understand the organization of the system.

A computer-based system (software is part of this system) exhibits one of the many available architectural styles. Every architectural style describes a system category that includes the following.

- *Computational components* such as clients, server, filter, and database to execute the desired system function
- A set of *connectors* such as procedure call, events broadcast, database protocols, and pipes to provide communication among the computational components
- *Constraints* to define integration of components to form a system

NOTES

- A *semantic* model, which enable the software designer to identify the characteristics of the system as a whole by studying the characteristics of its components.

Some of the commonly used architectural styles are *data-flow architecture*, *object-oriented architecture*, *layered system architecture*, *data-centered architecture*, and *call and return architecture*. Note that the use of an appropriate architectural style promotes design reuse, leads to code reuse, and supports interoperability.

Data-flow Architecture

Data-flow architecture is mainly used in the systems that accept some inputs and transform it into the desired outputs by applying a series of transformations. Each component, known as **filter**, transforms the data and sends this transformed data to other filters for further processing using the connector, known as **pipe** (see Figure 8.1(a)). Each filter works as an independent entity, that is, it is not concerned with the filter which is producing or consuming the data. A pipe is a unidirectional channel which transports the data received on one end to the other end. It does not change the data in anyway; it merely supplies the data to the filter on the receiver end.

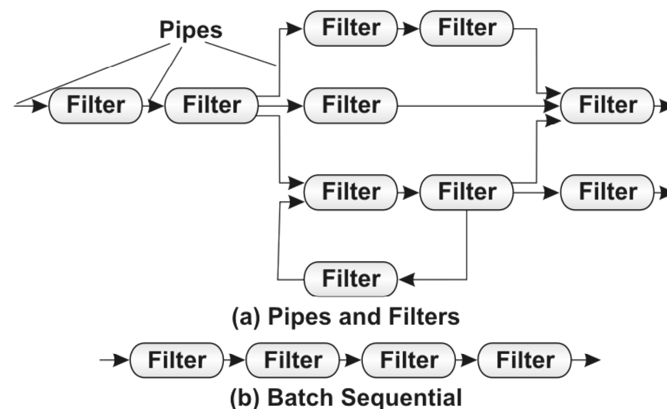


Fig. 8.1 *Data-flow Architecture*

Most of the times, the data-flow architecture degenerates a *batch sequential system* (see Figure 8.1(b)). In this system, a batch of data is accepted as input and then a series of sequential filters are applied to transform this data. One common example of this architecture is UNIX shell programs. In these programs, UNIX processes act as filters and the file system through which UNIX processes interact, act as pipes. Other well-known examples of this architecture are compilers, signal-processing systems, parallel programming, functional programming, and distributed systems. Some advantages associated with the data-flow architecture are listed below.

- It supports reusability.

- It is maintainable and modifiable.
- It supports concurrent execution.

Some disadvantages associated with the data-flow architecture are listed below.

NOTES

- It often degenerates to batch sequential system.
- It does not provide enough support for applications requires user interaction.
- It is difficult to synchronize two different but related streams.

Object-oriented Architecture

In object-oriented architectural style, components of a system encapsulate data and operations, which are applied to manipulate the data. In this style, components are represented as *objects* and they interact with each other through methods (connectors). This architectural style has two important characteristics, which are listed below.

- Objects maintain the integrity of the system.
- An object is not aware of the representation of other objects.

Some of the advantages associated with the object-oriented architecture are listed below.

- It allows designers to decompose a problem into a collection of independent objects.
- The implementation detail of objects is hidden from each other and hence, they can be changed without affecting other objects.

Layered Architecture

In layered architecture, several layers (components) are defined with each layer performing a well-defined set of operations. These layers are arranged in a hierarchical manner, each one built upon the one below it (see Figure 8.2). Each layer provides a set of services to the layer above it and acts as a client to the layer below it. The interaction between layers is provided through protocols (connectors) that define a set of rules to be followed during interaction. One common example of this architectural style is OSI-ISO (Open Systems Interconnection-International Organization for Standardization) communication system (see Figure 8.2).

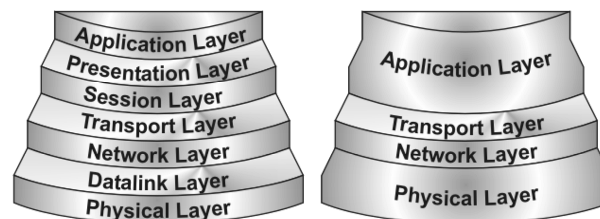


Fig. 8.2 *OSI and Internet Protocol Suite*

Data-centered Architecture

A data-centered architecture has two distinct components: a **central data structure** or data store (central repository) and a **collection of client software**. The data store (for example, a database or a file) represents the current state of the data and the client software performs several operations like add, delete, update, etc., on the data stored in the data store (see Figure 8.3). In some cases, the data store allows the client software to access the data independent of any changes or the actions of other client software.

In this architectural style, new components corresponding to clients can be added and existing components can be modified easily without taking into account other clients. This is because client components operate independently of one another.

A variation of this architectural style is blackboard system in which the data store is transformed into a *blackboard* that notifies the client software when the data (of their interest) changes. In addition, the information can be transferred among the clients through the *blackboard* component.

Some advantages of the data-centered architecture are listed below.

- Clients operate independently of one another.
- Data repository is independent of the clients.

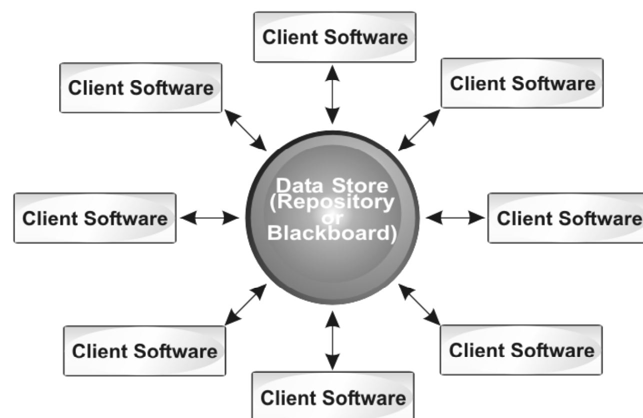


Fig. 8.3 *Data-centered Architecture*

- It adds scalability (that is, new clients can be added easily).
- It supports modifiability.
- It achieves data integration in component-based development using blackboard.

Call and Return Architecture

A call and return architecture enables software designers to achieve a program structure, which can be easily modified. This style consists of the following two substyles.

- **Main program/subprogram architecture:** In this, function is decomposed into a c'nents, which in turn may invoke other components (see Figure 8.4).

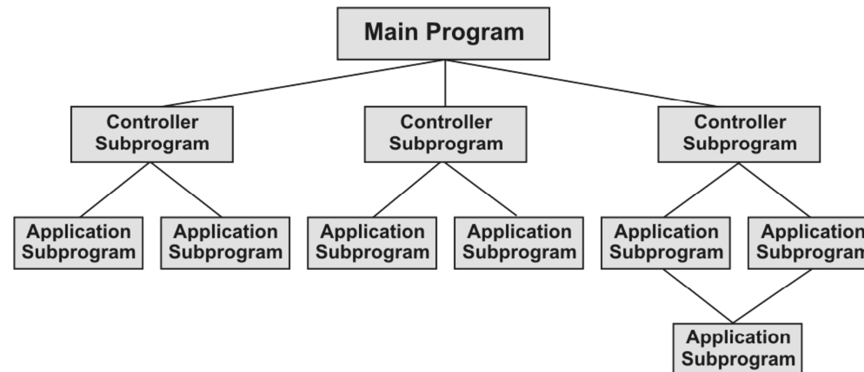


Fig. 8.4 *Main Program/Subprogram Architecture*

- **Remote procedure call architecture:** In this, components of the main or subprogram architecture are distributed over a network across multiple computers.

Component-Level Design

As soon as the first iteration of architectural design is complete, component-level design takes place. The objective of this design is to transform the design model into functional software. To achieve this objective, the component-level design represents the internal data structures and processing details of all the software components (defined during architectural design) at an abstraction level, closer to the actual code. In addition, it specifies an interface that may be used to access the functionality of all the software components.

The component-level design can be represented by using different approaches. One approach is to use a programming language while other is to use some intermediate design notation such as graphical (DFD, flowchart, or structure chart), tabular (decision table), or text-based (program design language) whichever is easier to be translated into source code.

The component-level design provides a way to determine whether the defined algorithms, data structures, and interfaces will work properly. Note that a component (also known as **module**) can be defined as a modular building block for the software. However, the meaning of component differs according to how software engineers use it. The modular design of the software should exhibit the following sets of properties.

- **Provide simple interface:** Simple interfaces decrease the number of interactions. Note that the number of interactions is taken into account while determining whether the software performs the desired function. Simple interfaces also provide support for reusability of components which reduces the cost to a greater extent. It not only decreases the time involved in design,

coding, and testing but the overall software development cost is also liquidated gradually with several projects. A number of studies so far have proven that the reusability of software design is the most valuable way of reducing the cost involved in software development.

- **Ensure information hiding:** The benefits of modularity cannot be achieved merely by decomposing a program into several modules; rather each module should be designed and developed in such a way that the information hiding is ensured. It implies that the implementation details of one module should not be visible to other modules of the program. The concept of information hiding helps in reducing the cost of subsequent design changes.

Modularity has become an accepted approach in every engineering discipline. With the introduction of modular design, complexity of software design has considerably reduced; change in the program is facilitated that has encouraged parallel development of systems. To achieve effective modularity, design concepts like functional independence are considered to be very important.

Functional Independence

Functional independence is the refined form of the design concepts of modularity, abstraction, and information hiding. Functional independence is achieved by developing a module in such a way that it uniquely performs given sets of function without interacting with other parts of the system. The software that uses the property of functional independence is easier to develop because its functions can be categorized in a systematic manner. Moreover, independent modules require less maintenance and testing activity, as secondary effects caused by design modification are limited with less propagation of errors. In short, it can be said that functional independence is the key to a good software design and a good design results in high-quality software. There exist two qualitative criteria for measuring functional independence, namely, *coupling* and *cohesion* (see Figure 8.5).

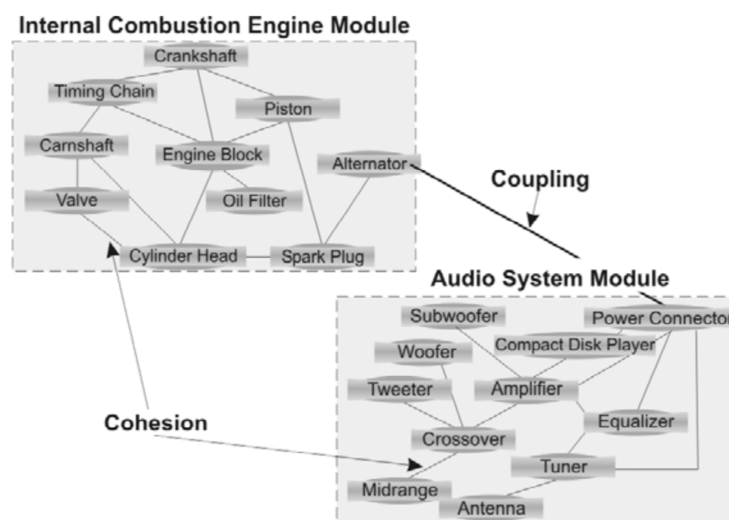


Fig. 8.5 *Coupling and Cohesion*

NOTES

Coupling

Coupling measures the degree of interdependence among the modules. Several factors like interface complexity, type of data that pass across the interface, type of communication, number of interfaces per module, etc. influence the strength of coupling between two modules. For better interface and well-structured system, the modules should be loosely coupled in order to minimize the 'ripple effect' in which modifications in one module results in errors in other modules. Module coupling is categorized into the following types.

- **No direct coupling:** Two modules are said to be *'no direct coupled'* if they are independent of each other. In Figure 8.6, Module 1 and Module 2 are 'not directly coupled'.

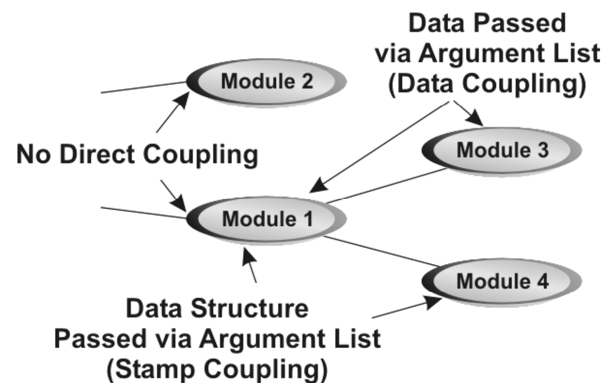


Fig. 8.6 *No Direct, Data, and Stamp Coupling*

- **Data coupling:** Two modules are said to be *'data coupled'* if they use parameter list to pass data items for communication. In Figure 8.6, Module 1 and Module 3 are data coupled.
- **Stamp coupling:** Two modules are said to be *'stamp coupled'* if they communicate by passing a data structure that stores additional information than what is required to perform their functions. In Figure 8.6, data structure is passed between Modules 1 and 4. Therefore, Module 1 and Module 4 are stamp coupled.
- **Control coupling:** Two modules are said to be '*control coupled*' if they communicate (pass a piece of information intended to control the internal logic) using at least one 'control flag'. The control flag is a variable whose value is used by the dependent modules to make decisions. In Figure 8.7, when Module 1 passes the control flag to Module 2, Module 1 and Module 2 are said to be control coupled.

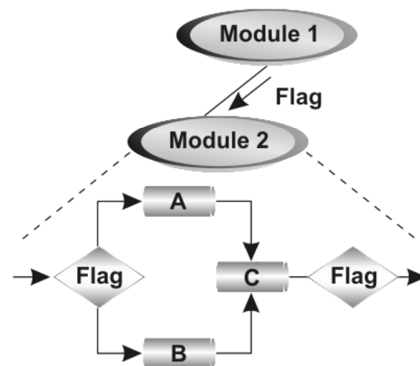


Fig. 8.7 Control Coupling

NOTES

- **Content coupling:** Two modules are said to be *'content coupled'* if one module modifies data of some other module or one module is under the control of another module or one module branches into the middle of another module. In Figure 8.8, Modules B and Module D are content coupled.

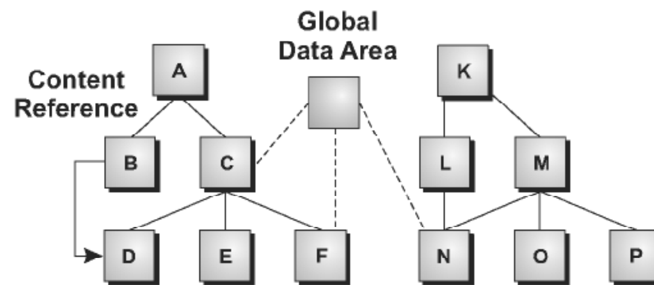


Fig. 8.8 Content and Common Coupling

- **Common coupling:** Two modules are said to be *'common coupled'* if they both reference a common data block. In Figure 8.8, Modules C and N are common coupled.

Cohesion

Cohesion measures the relative functional strength of a module. It represents the strength of bond between the internal elements of the modules. The tighter the elements are bound to each other, the higher will be the cohesion of a module. In practice, designers should avoid a low level of cohesion when designing a module. Generally, low coupling results in high cohesion and vice versa.

Various types of cohesion, as shown in Figure 8.9 are listed below.

- **Functional cohesion:** In this, the elements within the modules are concerned with the execution of a single function.
- **Sequential cohesion:** In this, the elements within the modules are involved in activities in such a way that the output from one activity becomes the input for the next activity.

NOTES

- **Communicational cohesion**: In this, the elements within the modules perform different functions, yet each function references the same input or output information.
- **Procedural cohesion:** In this, the elements within the modules are involved in different and possibly unrelated activities.
- **Temporal cohesion:** In this, the elements within the modules contain unrelated activities that can be carried out at the same time.

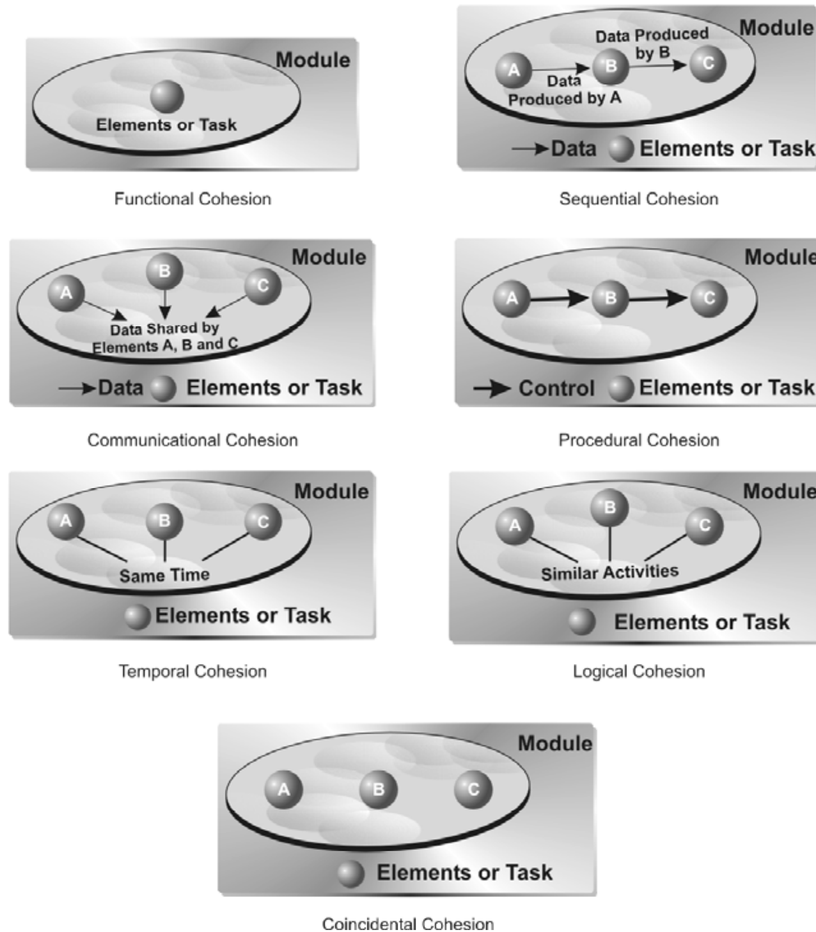


Fig. 8.9 *Types of Cohesion*

- **Logical cohesion:** In this, the elements within the modules perform similar activities, which are executed from outside the module.
- **Coincidental cohesion**: In this, the elements within the modules perform activities with no meaningful relationship to one another.

After having discussed various types of cohesions, Figure 8.10 illustrates the procedure which can be used in determining the types of module cohesion for software design.

NOTES

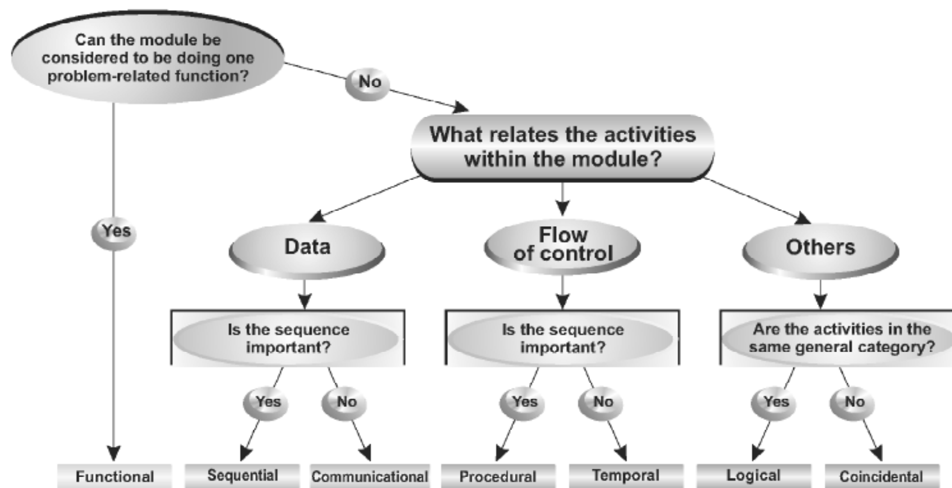


Fig. 8.10 *Selection Criteria of Cohesion*

Check Your Progress

1. What is data design?
2. Write a note on data center architecture?
3. Define coupling.
4. Define cohesion.

8.3 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Data design is the first design activity, which results in less complex, modular, and efficient program structure.A coupling is a device used to connect two shafts together at their ends for the purpose of transmitting power.
2. Data centered architecture is a layered process which provides architectural guidelines in data center development. Data Centered Architecture is also known as Database Centric Architecture.
3. Coupling measures the degree of interdependence among the modules. Several factors like interface complexity, type of data that pass across the interface, type of communication, number of interfaces per module, etc. influence the strength of coupling between two modules.
4. Cohesion measures the relative functional strength of a module. It represents the strength of bond between the internal elements of the modules.

8.4 SUMMARY

NOTES

- Data design helps to represent the data component in the conventional systems and class definitions in object-oriented systems.
- The process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.
- Architectural styles define a group of interlinked systems that share structural and semantic properties. In short, the objective of using architectural styles is to establish a structure for all the components present in a system.
- Data-flow architecture is mainly used in the systems that accept some inputs and transform it into the desired outputs by applying a series of transformations.
- Cohesion measures the relative functional strength of a module. It represents the strength of bond between the internal elements of the modules.
- A call and return architecture enables software designers to achieve a program structure, which can be easily modified.
- In object-oriented architectural style, components of a system encapsulate data and operations, which are applied to manipulate the data. In this style, components are represented as objects and they interact with each other through methods.
- Data-flow architecture is mainly used in the systems that accept some inputs and transform it into the desired outputs by applying a series of transformations.
- Architectural styles define a group of interlinked systems that share structural and semantic properties.

8.5 KEY WORDS

- **Dynamic Model:** This model specifies the behavioral aspect of the software architecture and indicates how the structure or system configuration changes as the function changes due to change in the external environment.
- **Process Model:** This model focuses on the design of the business or technical process, which must be implemented in the system.
- **Framework Model:** It Attempts to identify repeatable architectural design patterns encountered in similar types of application. This leads to an increase in the level of abstraction.

8.6 SELF ASSESSMENT QUESTIONS AND EXERCISES

NOTES

Short Answer Questions

1. What do you understand by data design?
2. What is architectural design document?
3. What are architectural styles?

Long Answer Questions

1. What is architectural design? Explain.
2. Explain the commonly used architectural styles.
3. Write a note on:
   (i) Functional independence
   (ii) Coupling
   (iii) cohesion

8.7 FURTHER READINGS

Pressman, Roger S. 1997. *Software Engineering, a Practitioner's Approach*. New Delhi: Tata McGraw-Hill.

Jalote, Pankaj. 1991. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

Schach, Stephen R. 2005. *Object Oriented and Classical Software Engineering*. New Delhi: Tata McGraw-Hill.

Somerville, Ian. 2001. *Software Engineering*. New Delhi: Pearson Education.

Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandriolli . 1991. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hill of India.

Jawadekar, Waman S. 2004. *Software Engineering: Principles and Practice*. New Delhi: Tata McGraw-Hill.

UNIT 9 USER INTERFACE DESIGN

NOTES

Structure

9.0 INTRODUCTION

In this unit, you will learn about the user interface design. It is the design of user interfaces for software and machine. The goal of designing is to make the interaction as simple and efficient as possible, in order to accomplish user goals.

9.1 OBJECTIVES

After going through this unit, you will be able to:

- Discuss the user interface design issues
- Understand the user interface rules
- Explain the steps for user interface design
- Understand evaluation techniques for user interface design

9.2 USER INTERFACE ANALYSIS AND DESIGN

User interfaces determine the way in which users interact with the software. The user interface design creates an effective communication medium between a human and a computing machine. It provides easy and intuitive access to information as well as efficient interaction and control of software functionality. For this, it is necessary for the designers to understand what the user requires from the user interface.

Since the software is developed for the user, the interface through which the user interacts with the software should also be given prime importance. The developer should interact with the person (user) for whom the interface is being

designed before designing it. Direct interaction with end-users helps the developers to improve the user interface design because it helps designers to know the user's goals, skills and needs. Figure 9.1 shows an example of a simple user interface design.

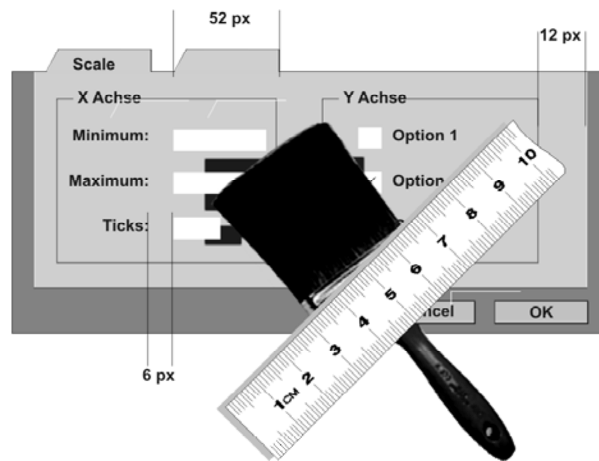


Fig. 9.1 An Example of a Simple User Interface Design

9.2.1 User Interface Design Issues

While designing a user interface, there are certain issues that must be considered. Generally, the software designers refer to these issues lately in the design process, which often results in project delays or customers' dissatisfaction. Therefore, it is recommended to consider these issues when the design process starts. Various design issues of user interface are listed below.

Response Time

The response time of a system is the time from when the user issues some request to the time the system responds to that request. The response time should be as low as possible, since longer response time results in user frustration. In addition, the response time for any specific request should not vary significantly. If this happens, user might think that something different has occurred. For example, it is not acceptable that a request sometimes responds in 0.5 seconds and sometimes in 2.5 seconds. Instead an average response time of 1 second is acceptable in this case.

Online Help System

Modern software applications provide an online help system that helps the users to learn about the operations of software at any time while using the software. In case the users forget a command or are unaware of some features of the software, they may invoke the online help system. The online help system must keep track of what the user is doing while invoking the help system so that it can provide help in the context of the user's actions. In addition, various other features that an online help system may incorporate in its design are listed below.

NOTES

- It may take advantage of graphics and animation characteristics while providing help to the users rather than representing just instructions.
- It may include help options for all or a subset of functions performed by the software.
- It may be invoked by the user using a help menu or help command or a special function key.
- It may use either a separate window or a reference to printed document or a fixed screen location in the same window for representing help messages.

Error Management

Sometimes during processing, errors occur due to some exceptional conditions such as out of memory, failure of communication link, etc. The system represents these errors to the users using error messages. These error messages are either unambiguous or contain only general information such as invalid input, system error, etc., which hardly provides any guidance to the user to correct errors. Thus, any error message or warning produced by the system should have the following characteristics.

- It should contain proper suggestions to recover from the errors. The users may be asked to invoke the online help system to find out more about the error situation.
- It should be represented in a language that is easily understandable by the user and should use polite words.
- It should mention the negative effects (if any) caused by the error encountered. For example, a file name that is corrupted because of that error.
- It should be associated with any audio or visual clue. For example, whenever an error occurs the system may generate a beep or change the text colour in order to indicate the occurrence of an error.

In general, users do not like errors irrespective of the fact how well designed it is. Thus, a good user interface should minimize the scope of committing errors. To manage the errors or to reduce their possibilities, various procedures like consistency of names, issue procedures, behaviour of similar commands, etc., can be used.

Menu and Command Labeling

Before GUI, the command language-based interfaces or menu-based interfaces were used for every type of application. The command language-based interface provides a set of primitive commands and a user can type an appropriate command to perform a particular function. Thus, a user is required to memorize a number of commands and type them in order to perform operations. On the other hand, in a menu-based interface, the user can select from a list of options (called menu) using

some pointing device like mouse. Thus, the user is relieved from the burden of remembering and typing commands. Several issues must be taken into consideration while designing a command language-based and menu-based user interface, which are listed below.

- Which menu options should be associated with a shortcut key?
- Which method is required for the user to execute any command? Several methods include pressing function keys, a pair of control keys (like Alt+D, Ctrl+S) or to type a word.
- How easy is it for the user to learn the commands and shortcut keys?
- Can a user assign a new shortcut key to any command or menu option?
- Are the menu labels self-explanatory?

9.2.2 User Interface Rules/Golden Rules

Designing a good and efficient user interface is a common objective among software designers. But what makes a user interface look 'good'? Software designers strive to achieve a good user interface by following three rules, namely, *ease of learning*, *efficiency of use*, and *aesthetic appeal*.

Ease of Learning

It determines the degree of ease with which a user can learn to interact with the software. This is especially important for novice users. However, on occasions, experienced users also face problems when using a new version of the software or while expanding their knowledge of using the software. Here, the *principle of state visualization* is applied, which states that whenever a change is made in the behaviour of the software, it should be reflected in the interface appearance.

While designing the user interface, designers should focus on the ease of learning for the class of users having little or no knowledge, since only few users have expertise in using all the features of the user interface. Generally, to ease the task of learning, designers make use of the following tools.

- **Affordance:** Provides hints about the features of a tool and the suggestions for using these features. For example, the style of the cap on some bottles indicates how to open it, that is, by rotating it clockwise or anti-clockwise. If no such indication is provided, one has to struggle while opening the bottle. Therefore, in this case, the bottle cap is not just a tool that physically helps to open the bottle, but it is also an affordance that shows how to open the bottle. Similarly, software designers should provide suggestions regarding the functioning of each part of the interface and how does it function.
- **Consistency:** Designers apply the *principle of coherence* to keep the interface consistent internally as well as externally. **Internal consistency** means behaviour of the program including minor details like colour, font, etc. should be uniform while using different parts of the program. For

NOTES

example, the colour of the pop-up menu should be uniform throughout the application. On the other hand, **external consistency** means the consistency of the program with the external environment. That is, the program should be consistent with the operating system in which it is running as well as with other applications that are running within that operating system.

Efficiency of Use

Once a user knows how to perform tasks, the next question is: how efficiently the interface can be used by a user to accomplish a task? Note that the efficiency of an interface can be determined only if the user is engaged in performing the tasks instead of learning how to perform.

To design an efficient interface, a thorough knowledge of behaviour of the users who are going to use the application is essential. For this, the designer should keep the following factors in mind.

- Frequency of using the interface
- The level of users (novice, intermediate, or expert)
- Frequency of performing a particular task.

In addition to these factors, the following guidelines can also help in designing an efficient interface.

- Users should be left to perform with the minimal physical actions required to accomplish a task. For this, the interface should provide the shortcut keys for performing the tasks that need to be performed frequently, since it reduces the work of users.
- Users should be left to perform with the minimal mental effort as well. For this, the interface should not assume the user to remember much detail; instead it should provide the necessary instructions for performing a task. For example, the interface should provide the file browser to select a file, instead of assuming the user to remember the path of the required file.

Aesthetically Pleasing

Today, look and feel is one of the biggest USPs (Unique Selling Points) while designing the software. A user always prefers to buy things that look attractive as well as make him feel better (as it provides ease of use) while using the product. Many software organizations focus specifically on designing the software, which has an attractive look and feel so that they can attract customers/users towards their product(s).

In addition to the before mentioned goals, there exists a *principle of metaphor* which, if applied to the software's design, results in a better and effective way of creating a user interface. This principle states that if the user interface for a complex software system is designed in such a way that it looks like the interface of an already existing system, it becomes easier to understand. For example, the

popular Windows operating system uses similar (not same) look and feel in all of its operating systems so that the users are able to use it in a user-friendly manner.

9.2.3 User Interface Design Process Steps

The user interface design, like all other software design elements, is an iterative process. Each step in this design occurs a number of times, each time elaborating and refining information developed in the previous step. Although many different user interface design models have been proposed, all have the following steps in common.

- **Analysis and modelling:** Initially, the profile of end-users for whom the interface is to be designed is analyzed, to determine their skill level and background. On this basis, users are classified into different categories, and for each category, the requirements are collected. After identifying the requirements, a detailed task analysis is made to identify the tasks that are to be performed by each class of users in order to achieve the system objectives. Finally, the environment in which user has to work is analyzed. Using the information gathered, an analysis model of the interface is designed.
- **Interface design:** Using the analysis model developed in the previous step, the interface object and the series of actions to accomplish the defined tasks are specified. In addition, a complete list of events (user actions), which causes the state of the user interface to change is also defined.
- **Interface implementation and validation:** A prototype of the interface is developed and is provided to the user to experiment with it. Once the user evaluates the prototype, it is modified according to their requirements. This process is repeated iteratively until all the requirements specified by the user are met.

To sum up, the user interface design activity starts with the identification of the user, task, and environmental requirements. After this, user states are created and analyzed to define a set of interface objects and actions. These objects then form the basis for the creation of screen layout, menus, icons, and much more. While designing the user interface, the following points must be considered.

- Follow the rules stated in further. If an interface does not follow any of these rules to a reasonable degree, then it needs to be redesigned.
- Determine how interface will be implemented.
- Consider the environment (like operating system, development tools, display properties, and so on).

9.2.4 Evaluating a User Interface Design

Although the interface design process results in a useful interface, it cannot be expected from a designer to design an interface of high quality in the first run. Each iteration of the steps involved in user interface design process leads to development

of a prototype. The objective of developing a prototype is to capture the 'essence' of the user interface. The prototype is evaluated, discrepancies are detected, and accordingly redesigning takes place. This process carries on until a good interface evolves.

Evaluating a user interface requires its prototype to include the look and feel of the actual interface and should offer a range of options. However, it is not essential for the prototype to support the whole range of software behaviour. Choosing an appropriate evaluation technique helps in knowing whether a prototype is able to achieve the desired user interface.

Evaluation Techniques

Evaluation of interface must be done in such a way that it can provide feedback to the next iteration. Each iteration of evaluation must specify what is good about the interface and where is a scope for improvement.

Some well known techniques for evaluating user interface are: *use it yourself*, *colleague evaluation*, *user testing*, and *heuristic evaluation*. Each technique has its own advantages and disadvantages and the emphasis of each technique for various issues like ease of learning and efficiency of use varies. The rule of aesthetic pleasing largely varies from user to user and can be evaluated well by observing what attracts the people.

- **Use it yourself:** This is the first technique of evaluation and in this technique the designer himself uses the interface for a period of time to determine its good and bad features. It helps the designer to remove the bad features of the interface. This technique also helps in identifying the missing components of the interface and its efficiency.
- **Colleague evaluation:** Since the designers are aware of the functionality of the software, it is possible that they may miss out on issues of ease of learning and efficiency. Showing the interface to a colleague may help in solving these issues. Note that if the prototype is not useful in the current state, colleagues might not spend sufficient time in using it to identify many efficiency issues.
- **User testing:** This testing is considered to be the most practical approach of evaluation. In this technique, users test the prototype with the expected differences recorded in the feedback. The communication between the users while testing the interface provides the most useful feedback. Before allowing the users to start testing, the designers choose some tasks expected to be performed by the users. In addition, they should prepare the necessary background details in advance. Users should spend sufficient time to understand these details before performing the test. This testing is considered the best way to evaluate ease of learning. However, it does not help much in identifying the inefficiency issues.

NOTES

- **Heuristic evaluation:** In this technique of evaluation, a checklist of issues is provided, which should be considered each time the user interface is evaluated. Such evaluations are inherently subjective in a sense that different evaluators find different set of problems in the interface. Usually, an experienced designer detects more errors than a less experienced designer. However, we cannot expect from a single person to detect all the errors. Therefore, it is always beneficial to employ multiple people in heuristic evaluation. The checklist provided in such evaluation includes the following issues.
  - o The interface should include as minimum extraneous information as possible.
  - o The interface should follow a natural flow and should lead the users' attention to the next expected action. For example, if the user is going to close an application without saving a task, then it must prompt the user to save or discard the changes.
  - o There should be a consistency in the use of color and other graphical elements. This helps the user to easily understand the information conveyed by the interface.
  - o The information should be presented to the user in an ordered manner and the amount of information should be as minimum as per required by the user.
  - o The messages should be displayed always from the user's perspective and not from the system's. For example, the messages like "you have purchased..." should be displayed rather than the messages like "we have sold you..."
  - o The interface should represent the sufficient information to the user whenever required, thereby minimizing the load on user's memory.
  - o The behaviour of the program should remain consistent internally as well as externally. For example, for representing similar items, same types of visual descriptions should be used.
  - o The user must be provided with a clear feedback about the actions of a program.
  - o The interface should provide shortcuts like hot keys, icons, etc. that help the users to speed up their operations as well as improve the effectiveness of a user interface.
  - o The error messages should be provided to communicate to the user that some error has occurred and these are the possible ways to correct it. These messages should describe exactly which part of the input has caused error.

If a user interface is carefully evaluated under these guidelines, almost all the problems related to learning and efficiency can be disclosed.

NOTES

Check Your Progress

1. What does user interface determines?
2. What are the three rules of user interface?
3. What are the well-known evaluation techniques?

9.3 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. User interfaces determine the way in which users interact with the software. The user interface design creates an effective communication medium between a human and a computing machine.
2. Software designers strive to achieve a good user interface by following three rules, namely, *ease of learning*, *efficiency of use*, and *aesthetic appeal*.
3. Some well-known techniques for evaluating user interface are: *use it yourself*, *colleague evaluation*, *user testing*, and *heuristic evaluation*.

9.4 SUMMARY

- User interfaces determine the way in which users interact with the software. The user interface design creates an effective communication medium between a human and a computing machine.
- The response time of a system is the time from when the user issues some request to the time the system responds to that request.
- Sometimes during processing, errors occur due to some exceptional conditions such as out of memory, failure of communication link, etc. The system represents these errors to the users using error messages.
- Designing a good and efficient user interface is a common objective among software designers.
- The user interface design, like all other software design elements, is an iterative process.
- Evaluation of interface must be done in such a way that it can provide feedback to the next iteration.
- It should contain proper suggestions to recover from the errors. The users may be asked to invoke the online help system to find out more about the error situation

9.5 KEY WORDS

NOTES

- **User Interfaces:** It determines the way in which users interact with the software.
- **Consistency:** Designers apply the principle of coherence to keep the interface consistent internally as well as externally.

9.6 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What do you understand by user interface design?
2. What do you understand by internal and external consistency?

Long Answer Questions

1. What are the user interface rules?
2. Explain the process of user interface design.
3. Explain the evaluation techniques of user interface design.

9.7 FURTHER READINGS

Pressman, Roger S. 1997. *Software Engineering, a Practitioner's Approach*. New Delhi: Tata McGraw-Hill.

Jalote, Pankaj. 1991. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

Schach, Stephen R. 2005. *Object Oriented and Classical Software Engineering*. New Delhi: Tata McGraw-Hill.

Somerville, Ian. 2001. *Software Engineering*. New Delhi: Pearson Education.

Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandriolli . 1991. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hill of India.

Jawadekar, Waman S. 2004. *Software Engineering: Principles and Practice*. New Delhi: Tata McGraw-Hill.

NOTES

BLOCK - IV
TESTING

UNIT 10 TESTING STRATEGIES

Structure

10.0 INTRODUCTION

Testing of software is critical, since testing determines the correctness, completeness and quality of the software being developed. Its main objective is to detect errors in the software. Errors prevent software from producing outputs according to user requirements. They occur if some part of the developed system is found to be incorrect, incomplete, or inconsistent. Errors can broadly be classified into three types, namely, requirements errors, design errors, and programming errors. To avoid these errors, it is necessary that: requirements are examined for conformance to user needs, software design is consistent with the requirements and notational convention, and the source code is examined for conformance to the requirements specification, design documentation and user expectations. All this can be accomplished through efficacious means of software testing.

NOTES

The activities involved in testing phase basically evaluate the capability of the developed system and ensure that the system meets the desired requirements. It should be noted that testing is fruitful only if it is performed in the correct manner. Through effective software testing, the software can be examined for correctness, comprehensiveness, consistency and adherence to standards. This helps in delivering high-quality software products and lowering maintenance costs, thus leading to more contented users.

10.1 OBJECTIVES

After going through this unit, you will be able to:

- Discuss the guidelines that are required to perform efficient and effective testing
- Design test plan, which specifies the purpose, scope, and method of software testing
- Understand various levels of testing including unit testing, integration testing, system testing, and acceptance testing
- Explain white box testing and black box testing techniques
- Explain how testing is performed in the object-oriented environment
- Explain how to perform debugging.

10.2 SOFTWARE TESTING FUNDAMENTALS

As already mentioned, software testing determines the correctness, completeness and quality of software being developed. IEEE defines testing as '*the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results.*'

Software testing is performed either manually or by using automated tools to make sure that the software is functioning in accordance with the user requirements. Various advantages associated with testing are listed below (also see Figure 10.1).

- It removes errors, which prevent software from producing outputs according to user requirements.
- It removes errors that lead to software failure.
- It ensures that the software conforms to business as well as user's needs.
- It ensures that the software is developed according to user requirements.
- It improves the quality of the software by removing maximum possible errors from it.

NOTES

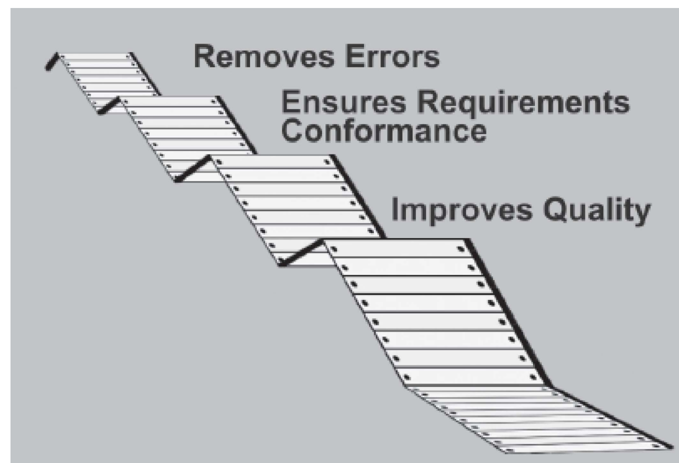


Fig. 10.1 *Advantages of Testing*

Bugs, error, fault and failure

Software testing is aimed at identifying any bugs, errors, faults, or failures (if any) present in the software. **Bug** is defined as a logical mistake, which is caused by a software developer while writing the software code. **Error** is defined as the measure of deviation of the outputs given by the software from the outputs expected by the user. **Fault** is defined as the condition that leads to malfunctioning of the software. Malfunctioning of software is caused due to several reasons such as change in the design, architecture or software code. Defect that causes error in operation or negative impact is called failure. **Failure** is defined as that state of software under which it is unable to perform functions according to user requirements. Bugs, errors, faults and failures prevents the software from performing efficiently and hence cause the software to produce unexpected outputs. Errors can be present in the software due to the following reasons.

- **Programming errors:** Programmers can make mistakes while developing the source code.
- **Unclear requirements:** The user is not clear about the desired requirements or the developers are unable to understand the user requirements in a clear and concise manner.
- **Software complexity:** The greater the complexity of the software, the more the scope of committing an error (especially by an inexperienced developer).
- **Changing requirements:** The users usually keep on changing their requirements, and it becomes difficult to handle such changes in the later stage of development process. Therefore, there are chances of making mistakes while incorporating these changes in the software.
- **Time pressures:** Maintaining schedule of software projects is difficult. When deadlines are not met, the attempt to speed up the work causes errors.

- **Poorly documented code:** If the code is not well documented or well written, then maintaining and modifying it becomes difficult. This causes errors to occur.

Note: In this unit, 'error' is used as a general term for 'bugs', 'errors', 'faults', and 'failures'.

NOTES

Guidelines of Software Testing

There are certain rules and guidelines that are followed during software testing. These guidelines act as a standard to test software and make testing more effective and efficient. The commonly used software testing guidelines are listed below (also see Figure 10.2).

- **Define the expected output:** When programs are executed during testing, they may or may not produce the expected outputs due to different types of errors present in the software. To avoid this, it is necessary to define the expected output before software testing begins. Without knowledge of the expected results, testers may fail to detect an erroneous output.
- **Inspect output of each test completely:** Software testing should be performed once the software is complete in order to check its performance and functionality along with occurrence of errors in various phases of software development.
- **Include test cases for invalid and unexpected conditions:** Generally, software produces correct outputs when it is tested using accurate inputs. However, if unexpected input is given to the software, it may produce erroneous outputs. Hence, test cases that detect errors even when unexpected and incorrect inputs are specified should be developed.
- **Test the modified program to check its expected performance:** Sometimes, when certain modifications are made in the software (like adding of new functions), it is possible that the software produces unexpected outputs. Hence, it should be tested to verify that it performs in the expected manner even after modifications.

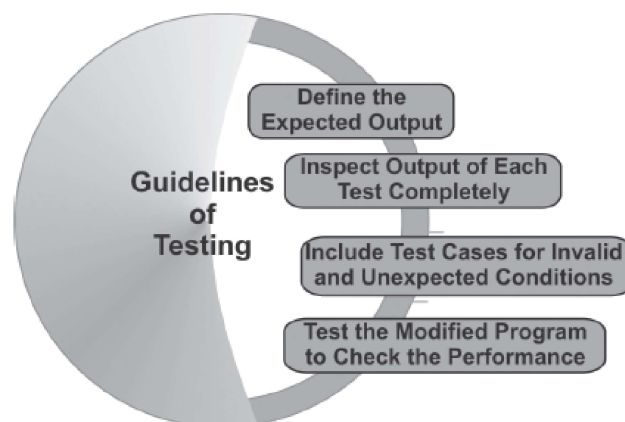


Fig. 10.2 *Software Testing Guidelines*

NOTES

Testability

The ease with which a program is tested is known as **testability**. Testability should always be considered while designing and implementing a software system so that the errors (if any) in the system can be detected with minimum effort. There are several characteristics of testability, which are listed below (also see Figure 10.3).

- **Easy to operate:** High-quality software can be tested in a better manner. This is because if software is designed and implemented considering quality, then comparatively fewer errors will be detected during the execution of tests.
- **Stability:** Software becomes stable when changes made to the software are controlled and when the existing tests can still be performed.
- **Observability:** Testers can easily identify whether the output generated for certain input is accurate simply by observing it.
- **Easy to understand:** Software that is easy to understand can be tested in an efficient manner. Software can be properly understood by gathering maximum information about it. For example, to have a proper knowledge of software, its documentation can be used, which provides complete information of software code thereby increasing its clarity and making testing easier.
- **Decomposability:** By breaking software into independent modules, problems can be easily isolated and the modules can be easily tested.

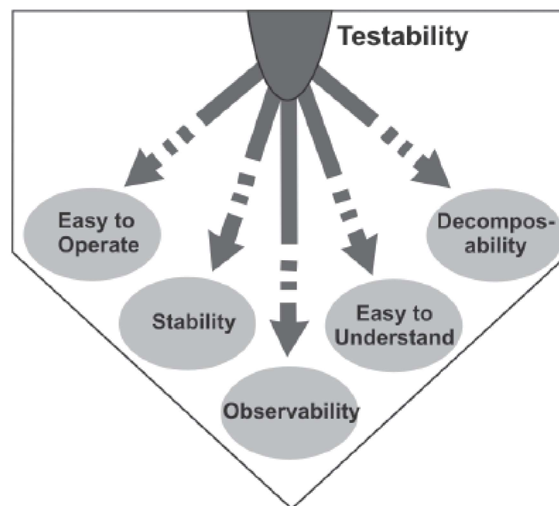


Fig. 10.3 *Testability*

Characteristics of Software Test

There are several tests (such as unit and integration) used for testing a software. Each test has its own characteristics. The following points, however, should be noted.

- **High probability of detecting errors:** To detect maximum errors, the tester should understand the software thoroughly and try to find the possible ways in which the software can fail. For example, in a program to divide two numbers, the possible way in which the program can fail is when 2 and 0 are given as inputs and 2 is to be divided by 0. In this case, a set of tests should be developed that can demonstrate an error in the division operator.
- **No redundancy:** Resources and testing time are limited in software development process. Thus, it is not beneficial to develop several tests, which have the same intended purpose. Every test should have a distinct purpose.
- **Choose the most appropriate test:** There can be different tests that have the same intent but due to certain limitations, such as time and resource constraint, only few of them are used. In such a case, the tests which are likely to find more errors should be considered.
- **Moderate:** A test is considered good if it is neither too simple nor too complex. Many tests can be combined to form one test case. However, this can increase the complexity and leave many errors undetected. Hence, all tests should be performed separately.

10.2.1 Test Plan

A test plan describes how testing would be accomplished. It is a document that specifies the purpose, scope, and method of software testing. It determines the testing tasks and the persons involved in executing those tasks, test items, and the features to be tested. It also describes the environment for testing and the test design and measurement techniques to be used. Note that a properly defined test plan is an agreement between testers and users describing the role of testing in software.

A complete test plan helps the people who are not involved in test group to understand why product validation is needed and how it is to be performed. However, if the test plan is not complete, it might not be possible to check how the software operates when installed on different operating systems or when used with other software. To avoid this problem, IEEE states some components that should be covered in a test plan. These components are listed in Table 10.1.

NOTES

Table 10.1 Components of a Test Plan

Component	Purpose
Responsibilities	Assigns responsibilities to different people and keeps them focused.
Assumptions	Avoids any misinterpretation of schedules.
Test	Provides an abstract of the entire process and outlines specific tests. The testing scope, schedule, and duration are also outlined.
Communication	Communication plan (who, what, when, how about the people) is developed.
Risk analysis	Identifies areas that are critical for success.
Defect reporting	Specifies the way in which a defect should be documented so that it may reoccur and be retested and fixed.
Environment	Describes the data, interfaces, work area, and the technical environment used in testing. All this is specified to reduce or eliminate the misunderstandings and sources of potential delay.

Test Case Design

A test case provides the description of inputs and their expected outputs to observe whether the software or a part of the software is working correctly. IEEE defines test case as '*a set of input values, execution preconditions, expected results and execution post conditions, developed for a particular objective or test condition such as to exercise a particular program path or to verify compliance with a specific requirement.*' Generally, a test case is associated with details like identifier, name, purpose, required inputs, test conditions, and expected outputs.

Incomplete and incorrect test cases lead to incorrect and erroneous test outputs. To avoid this, the test cases must be prepared in such a way that they check the software with all possible inputs. This process is known as **exhaustive testing** and the test case, which is able to perform exhaustive testing, is known as **ideal test case**. Generally, a test case is unable to perform exhaustive testing; therefore, a test case that gives satisfactory results is selected. In order to select a test case, certain questions should be addressed.

- How to select a test case?
- On what basis are certain elements of program included or excluded from a test case?

To provide an answer to these questions, test selection criterion is used that specifies the conditions to be met by a set of test cases designed for a given program. For example, if the criterion is to exercise all the control statements of a program at least once, then a set of test cases, which meets the specified condition should be selected.

10.2.2 Software Testing Strategies

NOTES

To perform testing in a planned and systematic manner, software testing strategy is developed. A **testing strategy** is used to identify the levels of testing which are to be applied along with the methods, techniques, and tools to be used during testing. This strategy also decides test cases, test specifications, test case decisions, and puts them together for execution.

Developing a test strategy, which efficiently meets the requirements of an organization, is critical to the success of software development in that organization. Therefore, a software testing strategy should contain complete information about the procedure to perform testing and the purpose and requirements of testing.

The choice of software testing strategy is highly dependent on the nature of the developed software. For example, if the software is highly data intensive then a strategy that checks structures and values properly to ensure that all inputs given to the software are correct and complete should be developed. Similarly, if it is transaction intensive then the strategy should be such that it is able to check the flow of all the transactions. The design and architecture of the software are also useful in choosing testing strategy.

A Strategic Approach to Software Testing

A number of software testing strategies are proposed for the testing process. All these strategies provide the tester a template, which is used for testing. Generally, all testing strategies have certain characteristics, which are listed below:

- Testing proceeds in an outward manner. It starts from testing the individual units, progresses to integrating these units, and finally, moves to system testing.
- Testing techniques used during different phases of software development are different.
- Testing is conducted by the software developer and by an ITG.
- Testing and debugging should not be used synonymously. However, any testing strategy must accommodate debugging with itself.

An efficient software testing strategy includes two types of tests, namely, *low-level tests* and *high-level tests*. **Low-level tests** ensure the correct implementation of small part of the source code and **high-level tests** ensure that major software functions are validated according to user requirements. A testing strategy sets certain milestones for the software such as final date for completion of testing and the date of delivering the software. These milestones are important when there is limited time to meet the deadline.

NOTES

Verification and Validation

Software testing is often used in association with the terms 'verification' and 'validation'. **Verification** refers to the process of ensuring that the software is developed according to its specifications. For verification, techniques like reviews, analysis, inspections and walkthroughs are commonly used. While **validation** refers to the process of checking that the developed software meets the requirements specified by the user. Verification and validation can be summarized thus as given here.

Verification: Is the software being developed in the right way?

Validation: Is the right software being developed?

Types of Software Testing Strategies

There are different types of software testing strategies, which are selected by the testers depending upon the nature and size of the software. The commonly used software testing strategies are listed below (also see Figure 10.4).

- **Analytic testing strategy:** This uses formal and informal techniques to access and prioritize risks that arise during software testing. It takes a complete overview of requirements, design, and implementation of objects to determine the motive of testing. In addition, it gathers complete information about the software, targets to be achieved, and the data required for testing the software.
- **Model-based testing strategy:** This strategy tests the functionality of software according to the real world scenario (like software functioning in an organization). It recognizes the domain of data and selects suitable test cases according to the probability of errors in that domain.
- **Methodical testing strategy:** It tests the functions and status of software according to the checklist, which is based on user requirements. This strategy is also used to test the functionality, reliability, usability, and performance of software.
- **Process-oriented testing strategy:** It tests the software according to already existing standards, such as IEEE standards. In addition, it checks the functionality of software by using automated testing tools.
- **Dynamic testing strategy:** This tests the software after having a collective decision of the testing team. Along with testing, this strategy provides information about the software, such as test cases used for testing the errors present in it.
- **Philosophical testing strategy:** It tests software assuming that any component of software can stop functioning anytime. It takes help from software developers, users and systems analysts to test the software.

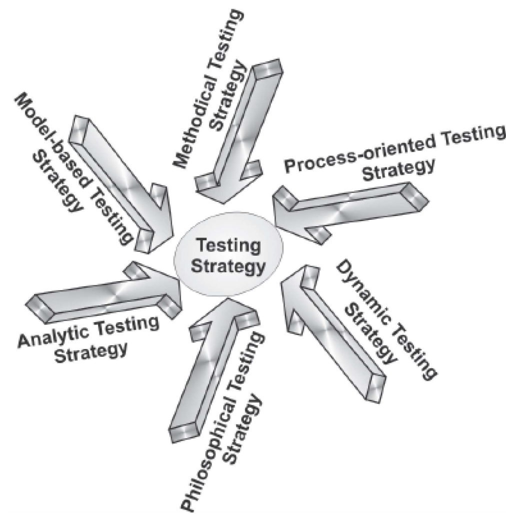


Fig. 10.4 *Types of Software Testing Strategy*

Developing a Software Testing Strategy

A testing strategy should be developed with intent to provide the most effective and efficient way of testing the software. While developing testing strategy, some questions arise, such as: when and what type of testing is to be done? What are the objectives of testing? Who is responsible for performing testing? And what outputs are produced as a result of testing? The inputs that should be available while developing testing strategy are listed below.

- Type of development project
- Complete information about the hardware and software components that are required to develop the software
- Risks involved
- Description of the resources that are required for the testing
- Description of all testing methods that are required to test various phases of SDLC
- Details of all the attributes that software is unable to provide. For example, software cannot describe its own limitations.

The output produced by the software testing strategy includes a detailed document, which indicates the entire test plan, including all test cases used during the testing phase. Testing strategy also specifies a list of testing issues that need to be resolved.

Organizing for Software Testing

Testing is an organizational issue, which is performed either by the *software developers* (who originally developed the software) or by an *independent test group* (*ITG*), which comprises software testers. The software developers are

NOTES

considered to be the best persons to perform testing as they have the best knowledge about the software. However, since software developers are involved in the development process, they may have their own interest to show that the software is error-free, meets user requirements, and is within schedule and budget. This vested interest hinders the process of testing.

To avoid this problem, the task of testing is assigned to an **independent test group** (**ITG**), which is responsible to detect errors that may have been neglected by the software developers. ITG tests the software without any discrimination since the group is not directly involved in the development process. However, the testing group does not completely take over the testing process; instead, it works with the software developers in the software project to ensure that testing is performed in an efficient manner. During the testing process, developers are responsible for correcting the errors uncovered by the testing group.

Generally, an ITG forms a part of the software development project team. This is because the group becomes involved during the specification activity and stays involved (planning and specifying test procedures) throughout the development process.

Various advantages and disadvantages associated with ITG are listed in Table 10.2.

Table 10.2 *Advantages and Disadvantages of ITG*

Advantages	Disadvantages
• ITG can more efficiently find defects related to interaction among different modules, system usability and performance, and many other special cases	• ITG may perform some tests that have already been performed by the developers. This results in duplication of effort as well as wastage of time.
• ITG serves the better solution than leaving testing to the developers. This is because the developers have neither training nor any motivation for testing.	• It is essential for the test group to be physically collocated with the design group; otherwise, problems may arise.
• Test groups can have better perception of how reliable is the software before delivering it to the user.	• Keeping a separate group for testing results in extra cost to the organization.

Note: Along with software testers, customers, end-users, and management also play an important role in software testing.

10.2.3 Levels of Testing

From a procedural point of view, software testing consists of a series of four steps (levels) that are performed sequentially. These steps are described as follows.

- **Unit testing:** This testing aims at testing the individual units of software. It makes heavy use of testing techniques that exercise specific control paths to detect errors in each software component individually.

- **Integration testing:** Once the individual units are tested, they are integrated and checked for interfaces between them. The integration testing focuses on issues associated with verification and program construction as components begin interacting with one another.
- **Validation testing:** This testing provides the assurance that the software constructed validates all the functional, behavioral, and performance requirements established during requirements analysis.
- **System testing:** This testing tests the entire software and the system elements as a whole. It ensures that the overall system functions according to the user requirements.

Strategic Issues

There are certain issues that need to be addressed for the successful implementation of software testing strategy. These issues are listed below.

- In addition to detecting errors, a good testing strategy should also assess portability and usability of the software.
- It should use quantifiable manner to specify software requirements such as outputs expected from software, test effectiveness, and mean time to failure which should be clearly stated in the test plan.
- It should improve testing method continuously to make it more effective.
- Test plans that support rapid cycle testing should be developed. The feedback from rapid cycle testing can be used to control the corresponding strategies.
- It should develop robust software, which is able to test itself using debugging techniques.
- It should conduct formal technical reviews to evaluate the test cases and test strategy. The formal technical reviews can detect errors and inconsistencies present in the testing process.

Test Strategies for Conventional Software

The test strategies chosen by most software teams for testing conventional software generally fall between two extremes. At one extreme is the unit testing where the individual components of the software are tested. While at the other extreme is the integration testing that facilitates the integration of components into a system and ends with tests that examine the integrated system.

10.2.4 Unit testing

Unit testing is performed to test the individual units of software. Since the software comprises various units/modules, detecting errors in these units is simple and consumes less time, as they are small in size. However, it is possible that the outputs produced by one unit become input for another unit. Hence, if incorrect output produced by one unit is provided as input to the second unit then it also

produces wrong output. If this process is not corrected, the entire software may produce unexpected outputs. To avoid this, all the units in the software are tested independently using unit testing (see Figure 10.5).

NOTES

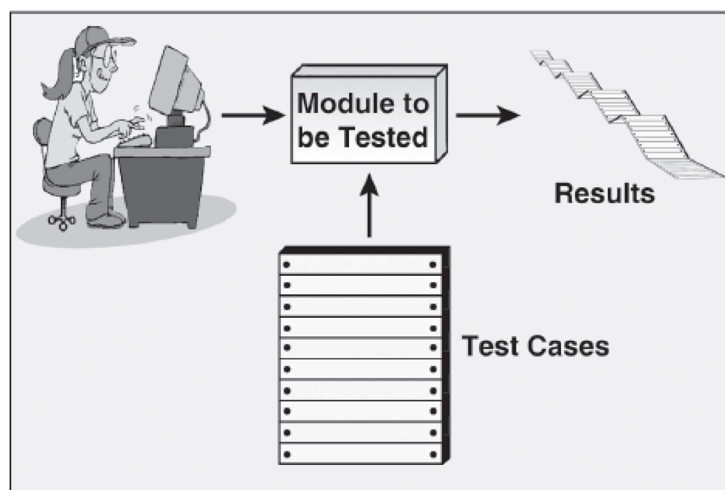


Fig. 10.5 *Unit Testing*

Unit testing is not just performed once during the software development, but repeated whenever the software is modified or used in a new environment. Some other points noted about unit testing are listed below.

- Each unit is tested separately regardless of other units of software.
- The developers themselves perform this testing.
- The methods of white box testing are used in this testing.

Unit testing is used to verify the code produced during software coding and is responsible for assessing the correctness of a particular unit of source code. In addition, unit testing performs the following functions.

- It tests all control paths to uncover maximum errors that occur during the execution of conditions present in the unit being tested.
- It ensures that all statements in the unit have been executed at least once.
- It tests data structures (like stacks, queues) that represent relationships among individual data elements.
- It checks the range of inputs given to units. This is because every input range has a maximum and minimum value and the input given should be within the range of these values.
- It ensures that the data entered in variables is of the same data type as defined in the unit.
- It checks all arithmetic calculations present in the unit with all possible combinations of input values.

Unit testing methods

Unit testing is performed by conducting a number of unit tests where each unit test checks an individual component that is either new or modified. A unit test is also referred to as a **module test** as it examines the individual units of code that constitute the program and eventually the software. In a conventional structured programming language such as C, the basic unit is a *function* or *subroutine* while in object-oriented language such as C++, the basic unit is a *class*.

Various tests that are performed as a part of unit testing are listed below (also see Figure 10.6).

- **Module interface:** This is tested to check whether information flows in a proper manner in and out of the 'unit' being tested. Note that test of data-flow (across a module interface) is required before any other test is initiated.
- **Local data structure:** This is tested to check whether temporarily stored data maintains its integrity while an algorithm is being executed.
- **Boundary conditions:** These are tested to check whether the module provides the desired functionality within the specified boundaries.
- **Independent paths:** These are tested to check whether all statements in a module are executed at least once. Note that in this testing, the entire control structure should be exercised.
- **Error-handling paths:** After successful completion of various tests, error-handling paths are tested.

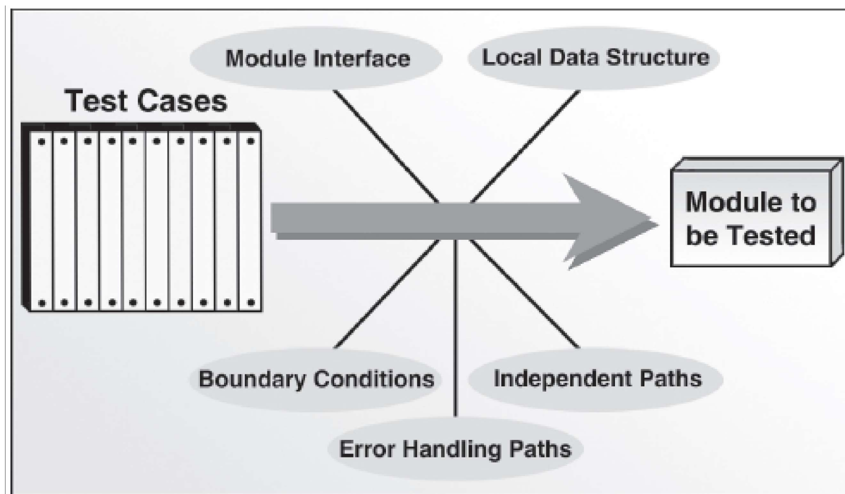


Fig. 10.6 *Unit Testing Methods*

Unit test case generation

Various unit test cases are generated to perform unit testing. Test cases are designed to uncover errors that occur due to erroneous computations, incorrect comparisons, and improper control flow. A proper unit test case ensures that unit testing is

performed efficiently. To develop test cases, the following points should always be considered.

NOTES

- **Expected functionality:** A test case is created for testing all functionalities present in the unit being tested. For example, an SQL query is given that creates `Table_A` and alters `Table_B`. Then a test case is developed to make sure that `Table_A` is created and `Table_B` is altered.
- **Input values:** Test cases are developed to check the various aspects of inputs, which are discussed here.
  - o **Every input value:** A test case is developed to check every input value, which is accepted by the unit being tested. For example, if a program is developed to print a table of five, then a test case is developed which verifies that only five is entered as input.
  - o **Validation of input:** Before executing software, it is important to verify whether all inputs are valid. For this purpose, a test case is developed which verifies the validation of all inputs. For example, if a numeric field accepts only positive values, then a test case is developed to verify that the numeric field is accepting only positive values.
  - o **Boundary conditions:** Generally, software fails at the boundaries of input domain (maximum and minimum value of the input domain). Thus, a test case is developed, which is capable of detecting errors that caused the software to fail at the boundaries of input domain. For example, errors may occur while processing the last element of an array. In this case, a test case is developed to check if an error occurs while processing the last element of the array.
  - o **Limitation of data types:** Variable that holds data types has certain limitations. For example, if a variable with data type long is executed then a test case is developed to ensure that the input entered for the variable is within the acceptable limit of long data type.
- **Output values:** A test case is designed to determine whether the desired output is produced by the unit. For example, when two numbers, '2' and '3' are entered as input in a program that multiplies two numbers, a test case is developed to verify that the program produces the correct output value, that is, '6'.
- **Path coverage:** There can be many conditions specified in a unit. For executing all these conditions, many paths have to be traversed. For example, when a unit consists of tested 'if' statements and all of them are to be executed, then a test case is developed to check whether all these paths are traversed.
- **Assumptions:** For a unit to execute properly, certain assumptions are made. Test cases are developed by considering these assumptions. For example,

NOTES

a test case is developed to determine whether the unit generates errors in case the assumptions are not met.

- **Abnormal terminations:** A test case is developed to check the behavior of a unit in case of abnormal termination. For example, when a power cut results in termination of a program due to shutting down of the computer, a test case is developed to check the behavior of a unit as a result of abnormal termination of the program.
- **Error messages:** Error messages that appear when the software is executed should be short, precise, self-explanatory, and free from grammatical mistakes. For example, if 'print' command is given when a printer is not installed, error message that appears should be 'Printer not installed' instead of 'Problem has occurred as no printer is installed and hence unable to print'. In this case, a test case is developed to check whether the error message displayed is according to the condition occurring in the software.

Unit testing procedure

Unit tests can be *designed* before coding begins or after the code is developed. Review of this design information guides the creation of test cases, which are used to detect errors in various units. Since a component is not an independent program, the drivers and/or stubs are used to test the units independently. **Driver** is a module that takes input from test case, passes this input to the unit to be tested and prints the output produced. **Stub** is a module that works as a unit referenced by the unit being tested. It uses the interface of the subordinate unit, does minimum data manipulation, and returns control back to the unit being tested (see Figure 10.7).

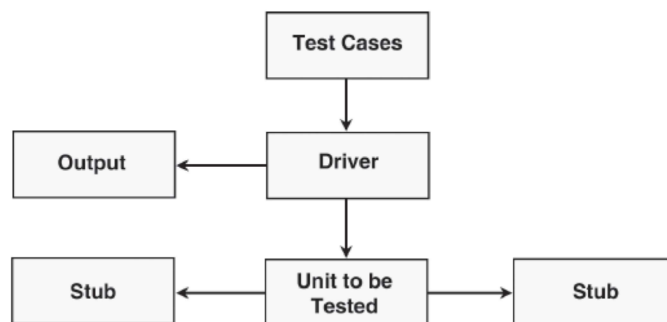


Fig. 10.7 *Unit Testing Environment*

Note: Drivers and stubs are not delivered with the final software product. Thus, they represent an overhead.

10.2.5 Integration Testing

Once unit testing is complete, integration testing begins. In integration testing, the units validated during unit testing are combined to form a subsystem. The integration testing is aimed at ensuring that all the modules work properly as per the user requirements when they are put together (that is, integrated).

NOTES

The objective of integration testing is to take all the tested individual modules, integrate them, test them again, and develop the software, which is according to design specifications (see Figure 10.8). Some other points that are noted about integration testing are listed below.

- It ensures that all modules work together properly and transfer accurate data across their interfaces.
- It is performed with an intention to uncover errors that lie in the interfaces among the integrated components.
- It tests those components that are new or have been modified or affected due to a change.

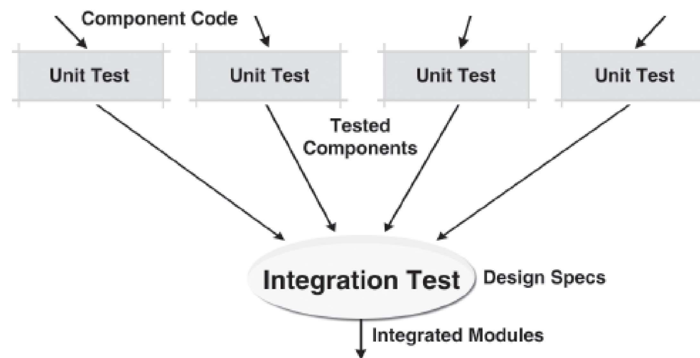


Fig. 10.8 *Integration of Individual Modules*

The *big bang* approach and *incremental integration* approach are used to integrate modules of a program. In the **big bang approach**, initially all modules are integrated and then the entire program is tested. However, when the entire program is tested, it is possible that a set of errors is detected. It is difficult to correct these errors since it is difficult to isolate the exact cause of the errors when the program is very large. In addition, when one set of errors is corrected, new sets of errors arise and this process continues indefinitely.

To overcome this problem, incremental integration is followed. The **incremental integration approach** tests program in small increments. It is easier to detect errors in this approach because only a small segment of software code is tested at a given instance of time. Moreover, interfaces can be tested completely if this approach is used. Various kinds of approaches are used for performing incremental integration testing, namely, *top-down integration testing*, *bottom-up integration testing*, *regression testing*, and *smoke testing*.

Top-down integration testing

In this testing, the software is developed and tested by integrating the individual modules, moving downwards in the control hierarchy. In top-down integration testing, initially only one module known as the main control module is tested.

After this, all the modules called by it are combined with it and tested. This process continues till all the modules in the software are integrated and tested.

It is also possible that a module being tested calls some of its subordinate modules. To simulate the activity of these subordinate modules, a stub is written. Stub replaces modules that are subordinate to the module being tested. Once the control is passed to the stub, it manipulates the data as least as possible, verifies the entry, and passes the control back to the module under test (see Figure 10.9). To perform top-down integration testing, the following steps are used.

1. The main control module is used as a test driver and all the modules that are directly subordinate to the main control module are replaced with stubs.
2. The subordinate stubs are then replaced with actual modules, one stub at a time. The way of replacing stubs with modules depends on the approach (depth first or breadth first) used for integration.
3. As each new module is integrated, tests are conducted.
4. After each set of tests is complete, its time to replace another stub with actual module.
5. In order to ensure no new errors have been introduced, regression testing may be performed.

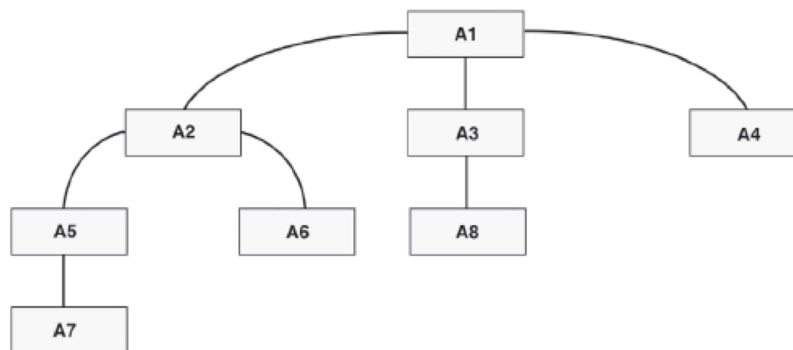
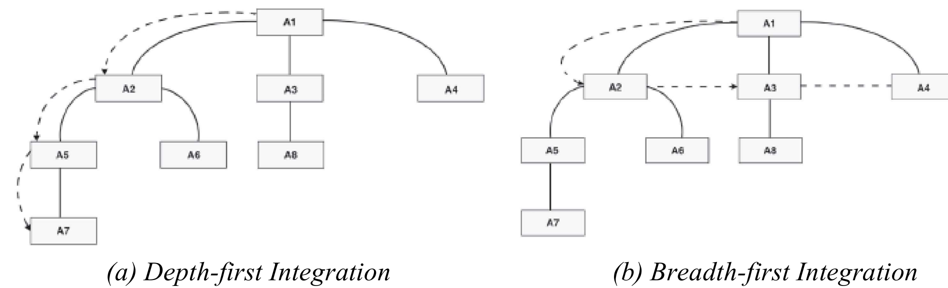


Fig. 10.9 *Top-down Integration Module*

Top-down integration testing uses either *depth-first integration* or *breadth-first integration* for integrating the modules. In **depth-first integration**, the modules are integrated starting from left and then move down in the control hierarchy. As shown in Figure 10.10 (a), initially, modules `A1`, `A2`, `A5` and `A7` are integrated. Then, module `A6` integrates with module `A2`. After this, control moves to the modules present at the centre of control hierarchy, that is, module `A3` integrates with module `A1` and then module `A8` integrates with module `A3`. Finally, the control moves towards right, integrating module `A4` with module `A1`.

NOTES



(a) Depth-first Integration (b) Breadth-first Integration

Fig. 10.10 *Top-down Integration*

In **breadth-first integration**, initially all modules at the first level are integrated moving downwards, integrating all modules at the next lower levels. As shown in Figure 10.10 b), initially modules `A2`, `A3`, and `A4` are integrated with module `A1` and then it moves down integrating modules `A5` and `A6` with module `A2` and module `A8` with module `A3`. Finally, module `A7` is integrated with module `A5.`

Various advantages and disadvantages associated with top-down integration are listed in Table 10.3.

Table 10.3 *Advantages and Disadvantages of Top-down Integration*

Advantages	Disadvantages
• Behavior of modules at high level is verified early. • None or only one driver is required. • Modules can be added one at a time with each step. • Supports both breadth-first method and depth-first method. • Modules are tested in isolation from other modules.	• Delays the verification of behavior of modules present at lower levels. • Large numbers of stubs are required in case the lowest level of software contains many functions. • Since stubs replace modules present at lower levels in the control hierarchy, no data flows upward in program structure. To avoid this, tester has to delay many tests until stubs are replaced with actual modules or has to integrate software from the bottom of the control hierarchy moving upward. • Module cannot be tested in isolation from other modules because it has to invoke other modules.

Bottom-up integration testing

In this testing, individual modules are integrated starting from the bottom and then moving upwards in the hierarchy. That is, bottom-up integration testing combines

and tests the modules present at the lower levels proceeding towards the modules present at higher levels of the control hierarchy.

Some of the low-level modules present in software are integrated to form clusters or builds (collection of modules). A test driver that coordinates the test case input and output is written and the clusters are tested. After the clusters have been tested, the test drivers are removed and the clusters are integrated, moving upwards in the control hierarchy.

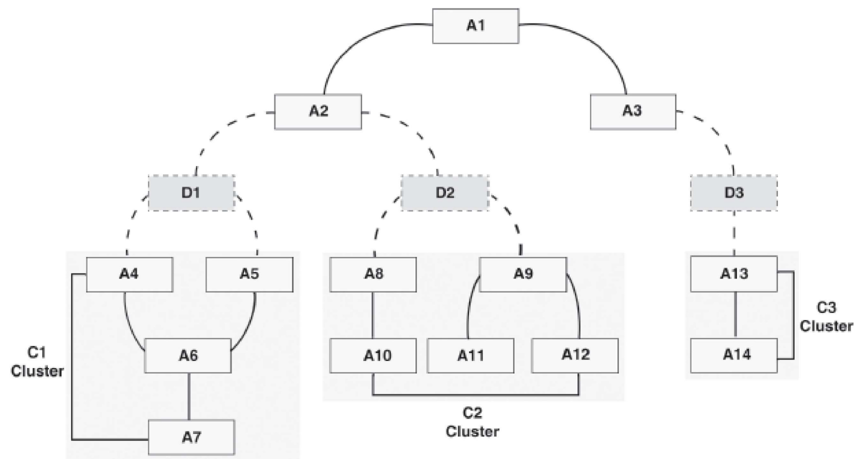


Fig. 10.11 *Bottom-up Integration*

Figure 10.11 shows modules, drivers, and clusters in bottom-up integration. The low-level modules A4, A5, A6, and A7 are combined to form cluster C1. Similarly, modules A8, A9, A10, A11, and A12 are combined to form cluster C2. Finally, modules A13 and A14 are combined to form cluster C3. After clusters are formed, drivers are developed to test these clusters. Drivers D1, D2, and D3 test clusters C1, C2, and C3 respectively. Once these clusters are tested, drivers are removed and clusters are integrated with the modules. Cluster C1 and cluster C2 are integrated with module A2. Similarly, cluster C3 is integrated with module A3. Finally, both the modules A2 and A3 are integrated with module A1.

Various advantages and disadvantages associated with bottom-up integration are listed in Table 10.4.

Table 10.4 *Advantages and Disadvantages of Bottom-up integration*

Advantages	Disadvantages
• Behavior of modules at lower levels is verified earlier. • No stubs are required. • Uncovers errors that occur at the lower levels in software. • Modules are tested in isolation from other modules.	• Delays in verification of modules at higher levels. • Large numbers of drivers are required to test clusters.

NOTES

Regression testing

Software undergoes changes every time a new module is integrated with the existing subsystem (Figure 10.12). Changes can occur in the control logic or input/output media, and so on. It is possible that new data-flow paths are established as a result of these changes, which may cause problems in the functioning of some parts of the software that was previously working perfectly. In addition, it is also possible that new errors may surface during the process of correcting existing errors. To avoid these problems, regression testing is used.

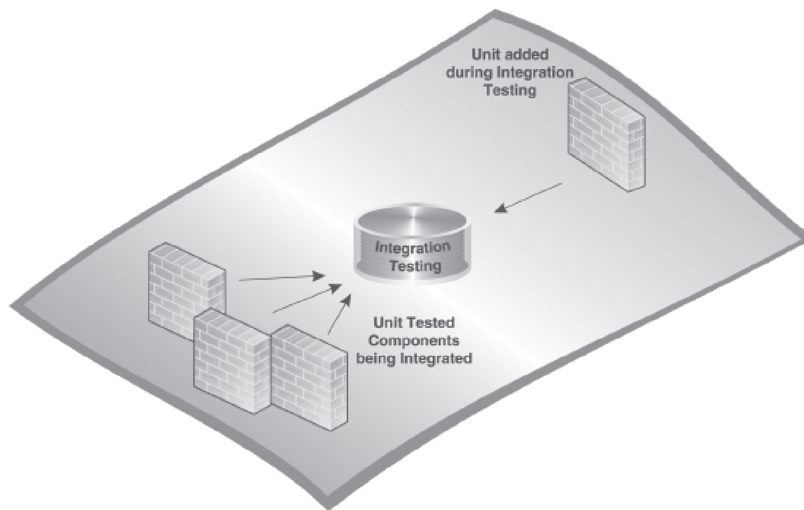


Fig. 10.12 *Addition of Module in Integration Testing*

Regression testing ‘re-tests’ the software or part of it to ensure that the components, features, and functions, which were previously working properly, do not fail as a result of the error correction process and integration of modules. It is regarded as an important activity as it helps in ensuring that changes (due to error correction or any other reason) do not result in additional errors or unexpected outputs from other system components.

To understand the need of regression testing, suppose an existing module has been modified or a new function is added to the software. These changes may result in errors in other modules of the software that were previously working properly. To illustrate this, consider the part of the code written below that is working properly.

```
x: = b + 1 ;
proc (z);
b: = x + 2; x: = 3;
```

Now suppose that in an attempt to optimize the code, it is transformed into the following.

```
proc(z);
b:= b + 3;
x:= 3;
```

This may result in an error if procedure `proc` accesses variable `x`. Thus, testing should be organized with the purpose of verifying possible degradations of correctness or other qualities due to later modifications. During regression testing, a subset of already defined test cases is re-executed on the changed software so that errors can be detected. Test cases for regression testing consist of three different types of tests, which are listed below.

- Tests that check all functions of the software
- Tests that check the functions that can be affected due to changes
- Tests that check the modified software modules.

The advantages and disadvantages associated with regression testing are listed in Table 10.5.

Table 10.5 *Advantages and Disadvantages of Regression Testing*

Advantages	Disadvantages
• Ensures that the unchanged parts of a software work properly. • Ensures that all errors that have occurred in the software due to modifications are corrected and are not affecting the working of software.	• Time consuming activity. • Considered to be expensive.

Smoke testing

Smoke testing is defined as an approach of integration testing in which a subset of test cases designed to check the main functionality of software are used to test whether the vital functions of the software work correctly. This testing is best suitable for testing time-critical software as it permits the testers to evaluate the software frequently.

Smoke testing is performed when the software is under development. As the modules of the software are developed, they are integrated to form a 'cluster'. After the cluster is formed, certain tests are designed to detect errors that prevent the cluster to perform its function. Next, the cluster is integrated with other clusters thereby leading to the development of the entire software, which is smoke tested frequently. A smoke test should possess the following characteristics.

- It should run quickly.
- It should try to cover a large part of the software and if possible the entire software.
- It should be easy for testers to perform smoke testing on the software.
- It should be able to detect all errors present in the cluster being tested.
- It should try to find showstopper errors.

NOTES

Generally, smoke testing is conducted every time a new cluster is developed and integrated with the existing cluster. Smoke testing takes minimum time to detect errors that occur due to integration of clusters. This reduces the risk associated with the occurrence of problems such as introduction of new errors in the software. A cluster cannot be sent for further testing unless smoke testing is performed on it. Thus, smoke testing determines whether the cluster is suitable to be sent for further testing. Other benefits associated with smoke testing are listed below.

- **Minimizes the risks, which are caused due to integration of different modules:** Since smoke testing is performed frequently on the software, it allows the testers to uncover errors as early as possible, thereby reducing the chance of causing severe impact on the schedule when there is delay in uncovering errors.
- **Improves quality of the final software:** Since smoke testing detects both functional and architectural errors as early as possible, they are corrected early, thereby resulting in a high-quality software.
- **Simplifies detection and correction of errors:** As smoke testing is performed almost every time a new code is added, it becomes clear that the probable cause of errors is the new code.
- **Assesses progress easily:** Since smoke testing is performed frequently, it keeps track of the continuous integration of modules, that is, the progress of software development. This boosts the morale of software developers.

Integration test documentation

To understand the overall procedure of software integration, a document known as **test specification** is prepared. This document provides information in the form of a test plan, test procedure, and actual test results.

1.0 Scope of Testing
2.0 Test Plan
    2.1 Test Phases and Builds
    2.2 Schedule
    2.3 Overhead Software
    2.4 Environment and Resources
3.0 Test Procedure 'n'
    3.1 Order of Integration
        3.1.1 Purpose
        3.1.2 Modules to be Tested
    3.2 Unit Tests for Modules in Build
    3.3 Description of Test for Module 'm'
    3.4 Overhead Software Description
    3.5 Expected Results
    3.6 Test Environment
    3.7 Special Tools or Techniques
    3.8 Test Case Data
    3.9 Expected Results for Build 'n'
4.0 Actual Test Results
5.0 References
6.0 Appendices

Fig. 10.13 *Test Specification Document*

NOTES

Figure 10.13 shows the test specification document, which comprises the following sections.

- **Scope of testing:** Outlines the specific design, functional, and performance characteristics of the software that need to be tested. In addition, it describes the completion criteria for each test phase and keeps track of the constraints that occur in the schedule.
- **Test plan:** Describes the testing strategy to be used for integrating the software. Testing is classified into two parts, namely, *phases* and *builds*. **Phases** describe distinct tasks that involve various subtasks. On the other hand, **builds** are groups of modules that correspond to each phase. Some of the common test phases that require integration testing include user interaction, data manipulation and analysis, display outputs, database management, and so on. Every test phase consists of a functional category within the software. Generally, these phases can be related to a specific domain within the architecture of the software. The criteria commonly considered for all test phases include interface integrity, functional validity, information content, and performance.

  In addition to test phases and builds, a test plan should also include the following.

  - A schedule for integration, which specifies the start and end date for each phase.
  - A description of overhead software that focuses on the characteristics for which extra effort may be required.
  - A description of the environment and resources required for the testing.
- **Test procedure 'n':** Describes the order of integration and the corresponding unit tests for modules. Order of integration provides information about the purpose and the modules that are to be tested. Unit tests are performed for the developed modules along with the description of tests for these modules. In addition, test procedure describes the development of overhead software, expected results during integration testing, and description of test case data. The test environment and tools or techniques used for testing are also mentioned in a test procedure.
- **Actual test results:** Provides information about actual test results and problems that are recorded in the test report. With the help of this information, it is easy to carry out software maintenance.
- **References:** Describes the list of references that are used for preparing user documentation. Generally, references include books and websites.
- **Appendices:** Provides information about the test specification document. Appendices serve as a supplementary material that is provided at the end of the document.

NOTES

Test Strategies for Object-Oriented Software

Like conventional software, the software testing in object-oriented (OO) software also aims to uncover maximum errors with minimum effort. However as the nature of object-oriented software is different from that of conventional software, the test strategies as well as testing techniques used for object-oriented software are also differ.

Unit testing in OO context

In object-oriented environment, the concept of unit is different. Here, the focus of unit testing is the class (or an instance of a class, usually called object), which is an encapsulated package binding the data and the operations that can be performed on these data together. But the smallest testable units in object-oriented software are the operations defined inside the class. Since in OO environment, a single operation may belong to many classes, it is ineffective to test any operation in a standalone fashion, as we do in conventional unit testing approach; rather an operation needs to be tested as a part of class. The class testing for object-oriented software is equivalent to the unit testing of conventional software. However, unlike unit testing of conventional software, it is not driven by the details of modules and data across module interfaces. Rather, it focuses on the operations defined inside the class and the state behavior of class.

Integration testing in OO context

The object-oriented software do not necessarily follow a hierarchical structure due to which the conventional top-down and bottom-up integration approaches are of little use for them. Moreover, conventional incremental integration approach (which means integrating operations one at a time into a class) also seems impossible because the operation being integrated into a class may need to interact directly or indirectly with other operations that form the class. To avoid such problems, two different integration approaches, including *thread-based testing* and *use-based testing*, are adopted for the integration testing of OO software.

In **thread-based testing** approach, the set of classes that need to respond an input or an event are determined. Each such set of classes is said to form a **thread**. After determining the sets of classes forming threads, each thread is integrated and tested individually. Regression testing is also performed to ensure that no error occur as a result of integration. On the other hand, in **use-based testing** approach, the integration process starts with the **independent classes** (the classes that either do not or do have a little collaboration with other classes). After the independent classes have been integrated and tested, the integration testing proceeds to next layer of classes called **dependent classes** which make use of independent classes. This integration procedure continues until the entire system has been integrated and tested.

Test Strategies for Web Applications

NOTES

The test strategy for Web applications (WebApps) conforms to the basic principles used for all software testing and follows the strategy and testing tactics recommended for object-oriented software. The steps followed in the strategic approach used for WebApps are summarized below.

1. Examine the content model for the WebApp to reveal errors.
2. Examine the interface model for the WebApp to ascertain whether all the use-cases can be conciliated.
3. Examine the design model for the WebApp to find out navigation errors.
4. Test the user interface to disclose errors in the presentation.
5. Perform unit testing for each functional component.
6. Test the navigation across the architecture.
7. Implement the WebApp in many different environmental configurations and check whether it is compatible with each environmental configuration.
8. Conduct the security testing with an aim to exploit vulnerabilities within the WebApp or in its environment.
9. Conduct the performance testing.
10. Make the WebApp tested by the end users and evaluate the results obtained from them for content and navigation errors, performance and reliability of WebApp, compatibility and usability concerns.

10.2.6 Validation Testing

After the individual components of a system have been unit tested, assembled as a complete system, and the interfacing errors have been detected as well as corrected, the validation testing begins. This testing is performed to ensure that the functional, behavioral and performance requirements of the software are met. IEEE defines validation testing as a '*formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the validation criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system*'.

During validation testing, the software is tested and evaluated by a group of users either at the developer's site or user's site. This enables the users to test the software themselves and analyze whether it is meeting their requirements. To perform acceptance testing, a predetermined set of data is given to the software as input. It is important to know the expected output before performing acceptance testing so that outputs produced by the software as a result of testing can be compared with them. Based on the results of tests, users decide whether to accept or reject the software. That is, if both outputs (expected and produced) match, the software is considered to be correct and is accepted; otherwise, it is rejected.

NOTES

The various advantages and disadvantages associated with validation testing are listed in Table 10.6.

Table 10.6 *Advantages and Disadvantages of Validation Testing*

Advantages	Disadvantages
• Gives user an opportunity to ensure that software meets user requirements, before actually accepting it from the developer. • Enables both users and software developers to identify and resolve problems in software. • Determines the readiness (state of being ready to operate) of software to perform operations. • Decreases the possibility of software failure to a large extent.	• The users may provide feedback without having proper knowledge of the software. • Since users are not professional testers, they may not able to either discover all software failures or accurately describe some failures.

Alpha and beta testing

Since the software is intended for large number of users, it is not possible to perform validation testing with all the users. Therefore, organizations engaged in software development use *alpha* and *beta* testing as a process to detect errors by allowing a limited number of users to test the software.

Alpha testing

Alpha testing is considered as a form of *internal* acceptance testing in which the users test the software at the *developer's* site (Figure 10.14). In other words, this testing assesses the performance of the software in the environment in which it is developed. On completion of alpha testing, users report the errors to software developers so that they can correct them.

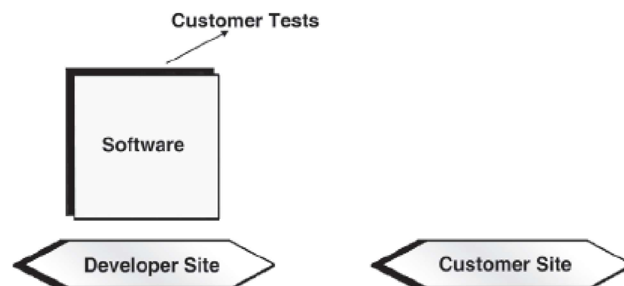


Fig. 10.14 *Alpha Testing*

Some advantages of alpha testing are listed below.

- It identifies all the errors present in the software.
- It checks whether all the functions mentioned in the requirements are implemented properly in the software.

NOTES

Beta testing

Beta testing assesses the performance of the software at *user's* site. This testing is 'live' testing and is conducted in an environment, which is not controlled by the developer. That is, this testing is performed without any interference from the developer (see Figure 10.15). Beta testing is performed to know whether the developed software satisfies the user requirements and fits within the business processes.

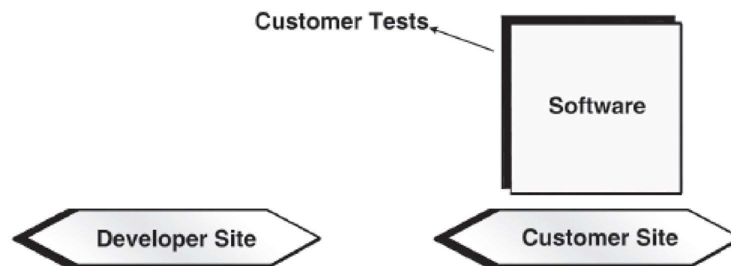


Fig. 10.15 *Beta Testing*

Note that beta testing is considered as *external* acceptance testing which aims to get feedback from the potential customers. For this, the system and the limited public tests (known as **beta versions**) are made available to the groups of people or the open public (for getting more feedback). These people test the software to detect any faults or bugs that may not have been detected by the developers and report their feedback. After acquiring the feedback, the system is modified and released either for sale or for further beta testing.

The advantages of beta testing are listed below.

- It evaluates the entire documentation of software. For example, it examines the detailed description of software code, which forms a part of documentation of software.
- It checks whether software is operating successfully in user environment or not.

10.2.7 System Testing

Software is integrated with other elements such as hardware, people, and database to form a computer-based system. This system is then checked for errors using system testing. IEEE defines system testing as '*a testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirement.*'

In system testing, the system is tested against non-functional requirements such as accuracy, reliability, and speed. The main purpose is to validate and verify the functional design specifications and to check how integrated modules work together. The system testing also evaluates the system's interfaces to other applications and utilities as well as the operating environment.

NOTES

During system testing, associations between objects (like fields), control and infrastructure, and the compatibility of the earlier released software versions with new versions are tested. System testing also tests some properties of the developed software, which are essential for users. These properties are listed below.

- **Usable:** Verifies that the developed software is easy to use and is understandable.
- **Secure:** Verifies that the access to important or sensitive data is restricted even for those individuals who have authority to use the software.
- **Compatible:** Verifies that the developed software works correctly in conjunction with existing data, software and procedures.
- **Documented:** Verifies that the manuals that give information about the developed software are complete, accurate and understandable.
- **Recoverable:** Verifies that there are adequate methods for recovery in case of failure.

System testing requires a series of tests to be conducted because software is only a component of computer-based system and finally it is to be integrated with other components such as information, people, and hardware. The test plan plays an important role in system testing as it describes the set of test cases to be executed, the order of performing different tests, and the required documentation for each test. During any test, if a defect or error is found, all the system tests that have already been executed must be re-executed after the repair has been made. This is required to ensure that the changes made during error correction do not lead to other problems.

While performing system testing, conformance tests and reviews can also be conducted to check the conformance of the application (in terms of interoperability, compliance, and portability) with corporate or industry standards.

System testing is considered to be complete when the outputs produced by the software and the outputs expected by the user are either in line or the difference between the two is within permissible range specified by the user. Various kinds of testing performed as a part of system testing (see Figure 10.16) are *recovery testing*, *security testing*, *stress testing*, and *performance testing*.

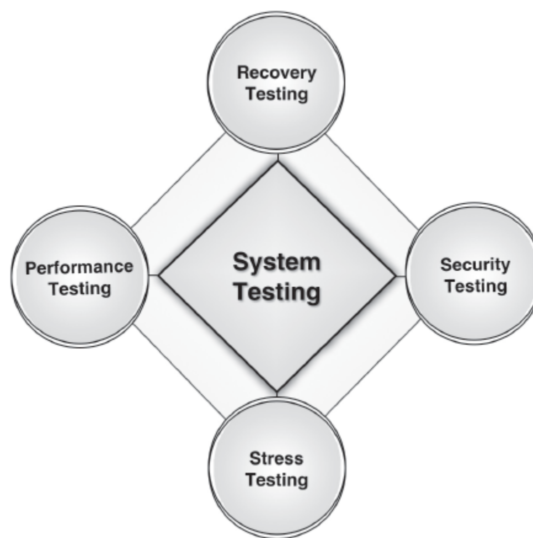


Fig. 10.16 *Types of System Testing*

Recovery testing

Recovery testing is a type of system testing in which the system is forced to fail in different ways to check whether the software recovers from the failures without any data loss. The events that lead to failure include system crashes, hardware failures, unexpected loss of communication, and other catastrophic problems.

To recover from any type of failure, a system should be fault-tolerant. A fault- tolerant system can be defined as a system, which continues to perform the intended functions even when errors are present in it. In case the system is not fault-tolerant, it needs to be corrected within a specified time limit after failure has occurred so that the software performs its functions in a desired manner.

Test cases generated for recovery testing not only show the presence of errors in a system, but also provide information about the data lost due to problems such as power failure and improper shutting down of computer system. Recovery testing also ensures that appropriate methods are used to restore the lost data. Other advantages of recovery testing are listed below.

- It checks whether the backup data is saved properly.
- It ensures that the backup data is stored in a secure location.
- It ensures that proper detail of recovery procedures is maintained.

Security testing

Systems with sensitive information are generally the target of improper or illegal use. Therefore, protection mechanisms are required to restrict unauthorized access to the system. To avoid any improper usage, security testing is performed which identifies and removes the flaws from software (if any) that can be exploited by the

NOTES

intruders and thus, result in security violations. To find such kind of flaws, the tester like an intruder tries to penetrate the system by performing tasks such as cracking the password, attacking the system with custom software, intentionally producing errors in the system, etc. The security testing focuses on the following areas of security.

- **Application security:** To check whether the user can access only those data and functions for which the system developer or user of system has given permission. This security is referred to as **authorization**.
- **System security:** To check whether only the users, who have permission to access the system, are accessing it. This security is referred to as **authentication**.

Generally, the disgruntled/dishonest employees or other individuals outside the organization make an attempt to gain unauthorized access to the system. If such people succeed in gaining access to the system, there is a possibility that a large amount of important data can be lost resulting in huge loss to the organization or individuals.

Security testing verifies that the system accomplishes all the security requirements and validates the effectiveness of these security measures. Other advantages associated with security testing are listed below.

- It determines whether proper techniques are used to identify security risks.
- It verifies that appropriate protection techniques are followed to secure the system.
- It ensures that the system is able to protect its data and maintain its functionality.
- It conducts tests to ensure that the implemented security measures are working properly.

Stress testing

Stress testing is designed to determine the behavior of the software under abnormal situations. In this testing, the test cases are designed to execute the system in such a way that abnormal conditions arise. Some examples of test cases that may be designed for stress testing are listed below.

- Test cases that generate interrupts at a much higher rate than the average rate
- Test cases that demand excessive use of memory as well as other resources
- Test cases that cause 'thrashing' by causing excessive disk accessing.

IEEE defines stress testing as '*testing conducted to evaluate a system or component at or beyond the limits of its specified requirements*.' For example, if a software system is developed to execute 100 statements at a time, then stress testing may generate 110 statements to be executed. This load may increase until the software fails. Thus, stress testing specifies the way in which a system reacts

when it is made to operate beyond its performance and capacity limits. Some other advantages associated with stress testing are listed below.

- It indicates the expected behavior of a system when it reaches the extreme level of its capacity.
- It executes a system till it fails. This enables the testers to determine the difference between the expected operating conditions and the failure conditions.
- It determines the part of a system that leads to errors.
- It determines the amount of load that causes a system to fail.
- It evaluates a system at or beyond its specified limits of performance.

Performance testing

Performance testing is designed to determine the performance of the software (especially real-time and embedded systems) at the run-time in the context of the entire computer-based system. It takes various performance factors like load, volume, and response time of the system into consideration and ensures that they are in accordance with the specifications. It also determines and informs the software developer about the current performance of the software under various parameters (like condition to complete the software within a specified time limit).

Often performance tests and stress tests are used together and require both software and hardware instrumentation of the system. By instrumenting a system, the tester can reveal conditions that may result in performance degradation or even failure of a system. While the performance tests are designed to assess the throughput, memory usage, response time, execution time, and device utilization of a system, the stress tests are designed to assess its robustness and error handling capabilities. Performance testing is used to test several factors that play an important role in improving the overall performance of the system. Some of these factors are listed below.

- **Speed:** Refers to the extent how quickly a system is able to respond to its users. Performance testing verifies whether the response is quick enough.
- **Scalability:** Refers to the extent to which the system is able to handle the load given to it. Performance testing verifies whether the system is able to handle the load expected by users.
- **Stability:** Refers to the extent how long the system is able to prevent itself from failure. Performance testing verifies whether the system remains stable under expected and unexpected loads.

The outputs produced during performance testing are provided to the system developer. Based on these outputs, the developer makes changes to the system in order to remove the errors. This testing also checks the system characteristics such as its reliability. Other advantages associated with performance testing are listed below.

- It assess whether a component or system complies with specified performance requirements.
- It compares different systems to determine which system performs better.

NOTES

Check Your Progress

1. What does software testing determines?
2. What is testability?
3. What is a test plan?

10.3 TESTING CONVENTIONAL APPLICATIONS

As stated earlier, after the software has been developed, it should be tested in a proper manner before delivering it to the user. To test any software, test cases need to be designed. There are two techniques that provide systematic guidance for designing tests cases for conventional applications. These techniques are:

- Once the internal working of software is known, tests are performed to ensure that all internal operations of software are performed according to specifications. This is referred to as **white box testing**.
- Once the specified function for which software has been designed is known, tests are performed to ensure that each function is working properly. This is referred to as **black box testing**.

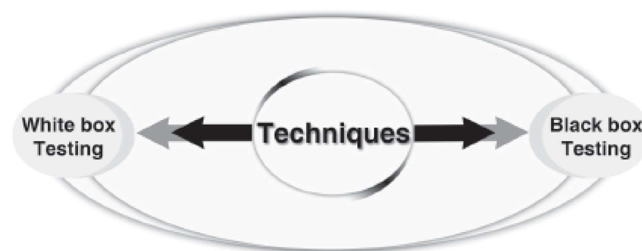


Fig. 10.17 *Testing Techniques*

10.3.1 White Box Testing

White box testing, also known as **structural testing** or **glass box testing**, is performed to check the internal structure of a program. To perform white box testing, the tester should have a thorough knowledge of the program internals along with the purpose of developing the software. During this testing, the entire software implementation is also included with the specification. This helps in detecting errors even with unclear or incomplete software specification.

The goal of white box testing is to ensure that the test cases (developed by software testers by using white box testing) exercise each path through a program. That is, test cases ensure that all internal structures in the program are developed

NOTES

according to design specifications (see Figure 10.18). The test cases also ensure the following.

- All independent paths within the program have been exercised at least once.
- All internal data structures have been exercised.
- All loops (simple loops, concatenated loops, and nested loops) have been executed at and within their specific boundaries.
- All segments present between the controls structures (like 'switch' statement) have been executed at least once.
- Each branch (like 'case' statement) has been exercised at least once.
- All the logical conditions as well as their combinations have been executed at least once for both true and false paths.

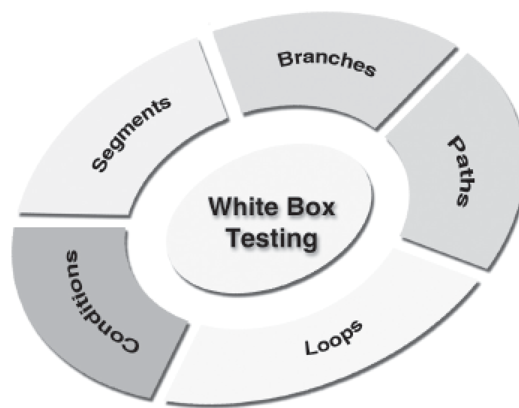


Fig. 10.18 *White Box Testing*

The various advantages and disadvantages of white box testing are listed in Table 10.7.

Table 10.7 *Advantages and Disadvantages of White Box Testing*

Advantages	Disadvantages
• Covers the larger part of the program code while testing. • Uncovers typographical errors. • Detects design errors that occur when incorrect assumptions are made about execution paths	• Tests that cover most of the program code may not be good for assessing the functionality of surprise (unexpected) behaviors and other testing goals. • Tests based on design may miss other system problems. • Tests cases need to be changed if implementation changes.

The effectiveness of white box testing is commonly expressed in terms of test or code coverage metrics, that is, the fraction of code exercised by test cases. Various types of testing, which occur as part of white box testing are *basis path testing*, *control structure testing*, and *mutation testing* (see Figure 10.19).

NOTES

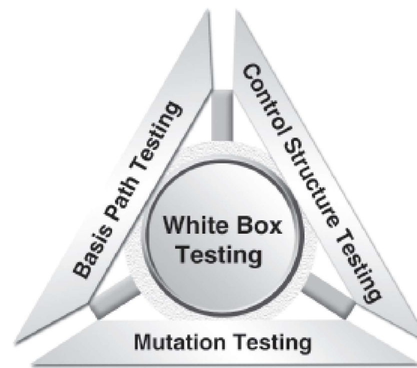


Fig. 10.19 Types of White Box Testing

Basis path testing

Basis path testing enables to generate test cases such that every path of the program has been exercised at least once. This technique is used to specify the basic set of execution paths that are required to execute all the statements present in the program. Note that with the increase in the size of the software the number of execution paths also increases thereby degrading the effectiveness of basis path testing.

Creating flow graph

A flow graph represents the logical control flow within a program. For this, it makes use of a notation as shown in Figure 10.20.

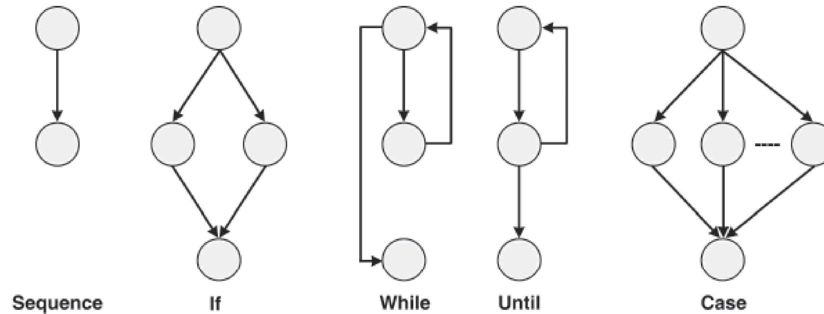
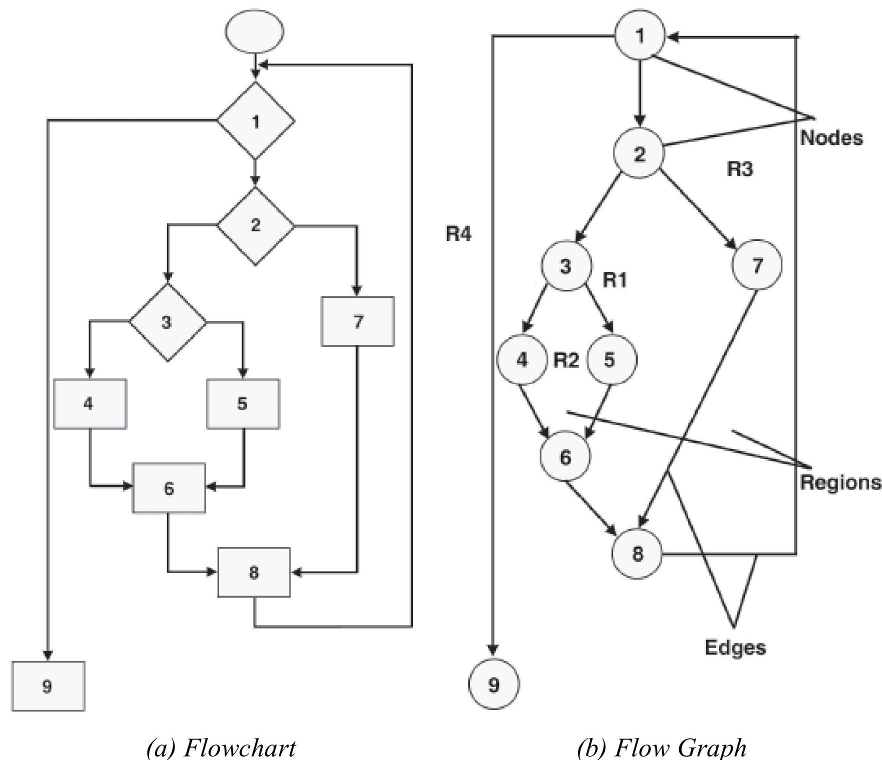


Fig. 10.20 Flow Graph Notation

A flow graph uses different symbols, namely, *circles* and *arrows* to represent various statements and flow of control within the program. Circles represent **nodes**, which are used to depict the procedural statements present in the program. A sequence of process boxes and a decision box used in a flowchart can be easily mapped into a single node. Arrows represent **edges** or **links**, which are used to depict the flow of control within the program. It is necessary for every edge to end in a node irrespective of whether it represents a procedural statement. In a flow graph, the area bounded by edges and nodes is known as a **region**. In addition, the area outside the graph is also counted as a region while counting regions. A flow graph can be easily understood with the help of a diagram. For example, in

Figure 10.21 (a) a flowchart has been depicted, which has been represented as a flow graph in 10.21 (b).



(a) Flowchart *(b) Flow Graph*

Fig. 10.21 *Creating Flow Graph*

Note that a node that contains a condition is known as **predicated node**, which contains one or more edges emerging out of it. For example, in Figure 10.21(b), node 2 and node 3 represent the predicated nodes.

Finding independent paths

A path through the program, which specifies a new condition or a minimum of one new set of processing statements, is known as an independent path. For example, in nested 'if' statements there are several conditions that represent independent paths. Note that a set of all independent paths present in the program is known as a **basis set**.

A test case is developed to ensure that while testing all statements of the program get exercised at least once. For example, all the independent paths in Figure 10.21(b) are listed below.

```
P1: 1-9
P2: 1-2-7-8-1-9
P3: 1-2-3-4-6-8-1-9
P4: 1-2-3-5-6-8-1-9
```

NOTES

Where P1, P2, P3, and P4 represent different independent paths present in the program.

To determine the number of independent paths through a program, the **cyclomatic complexity metric** is used that provides a quantitative measure of the logical complexity of a program. The value of this metric defines the number of test cases that should be developed to ensure that all statements in the program get exercised at least once during testing.

Cyclomatic complexity of a program can be computed by using any of the following three methods.

- By counting the total number of regions in the flow graph of a program. For example, in Figure 10.21 (b), there are four regions represented by R1, R2, R3, and R4; hence, the cyclomatic complexity is four.
- By using the following formula.

  ```
  CC = E - N + 2
  ```

  Where

  CC = the cyclomatic complexity of the program

  E = the number of edges in the flow graph

  N = the number of nodes in the flow graph.

  For example, in Figure 10.21 (b), E = 11, N = 9. Therefore, CC = 11 - 9 + 2 = 4.
- By using the following formula.

  ```
  CC = P + 1
  ```

  Where

  P = the number of predicate nodes in the flow graph.

  For example, in Figure 10.21 (b), P = 3. Therefore, CC = 3 + 1 = 4.

Note: Cyclomatic complexity can be calculated either manually (generally for small program suites) or using automated tools. However, for most operational environments, automated tools are preferred.

Deriving test cases

In this, basis path testing is presented as a series of steps and the test cases are developed to ensure that all statements within the program get exercised at least once while performing testing. While performing basis path testing, initially the basis set (independent paths in the program) is derived. The basis set can be derived using the steps listed below.

1. **Draw the flow graph of the program:** A flow graph is constructed using symbols previously discussed. For example, a program to find the greater of two numbers is given below.

   ```
   procedure greater;
   integer: x, y, z = 0;
   ```

NOTES

```
1   enter the value of x;
2   enter the value of y;
3   if x > y then
4      z = x;
    else
5      z = y;
6   end greater
```

The flow graph for the above program is shown in Figure 10.22.

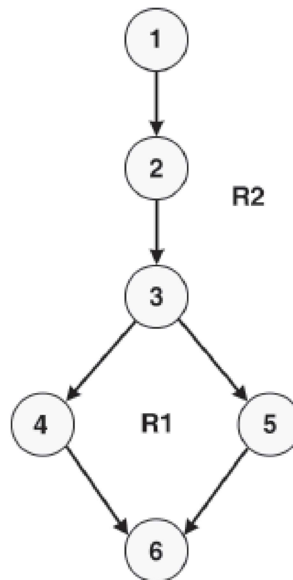


Fig. 10.22 Flow Graph to Find the Greater between Two Numbers

2. **Compute the cyclomatic complexity:** The cyclomatic complexity of the program can be computed using the flow graph depicted in Figure 10.22 as given below.

   `CC` = 2 as there are two regions `R1` and `R2`

   or

   `CC` = 6 edges – 6 nodes + 2 = 2

   or

   `CC` = 1 predicate node + 1 = 2

3. **Determine all independent paths through the program:** For the flow graph depicted in Figure 10.22, the independent paths are listed below.

   `P1:  1-2-3-4-6`
   `P2:  1-2-3-5-6`

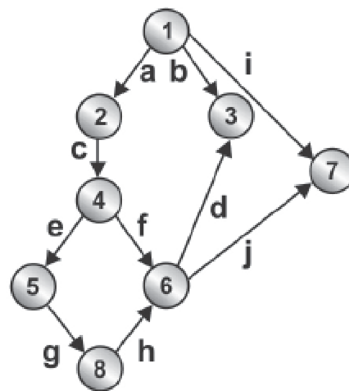
4. **Prepare test cases:** Test cases are prepared to implement the execution of all independent paths in the basis set. The program is then tested for each test case and the produced output is compared with the desired output.

NOTES

Generating graph matrix

Graph matrix is used to develop software tool that in turn helps in carrying out basis path testing. It is defined as a data structure used to represent the flow graph of a program in a tabular form. This matrix is also used to evaluate the control structures present in the program during testing.

Graph matrix is a square matrix of the size `N×N`, where `N` is the number of nodes in the flow graph. An entry is made in the matrix at the intersection of `i`th row and `j`th column if there exists an edge between `i`th and `j`th node in the flow graph. Every entry in the graph matrix is assigned some value known as **link weight**. Adding link weights to each entry makes the graph matrix a useful tool for evaluating the control structure of the program during testing. Figure 10.23 (b) shows the graph matrix generated for the flow graph depicted in Figure 10.23 (a).



(a) Flow Graph

	1	2	3	4	5	6	7	8
1		a	b				i	
2				c				
3								
4					e	f		
5								g
6			d				j	
7								
8						h		

(b) Graph Matrix

Fig. 10.23 *Generating Graph Matrix*

In the flow graph shown in Figure 10.23 (a), numbers and letters are used to identify each node and edge respectively. In Figure 10.23 (b), a letter entry is made if there is an edge between two nodes of the flow graph. For example, node 3 is connected to the node 6 by edge `d` and node 4 is connected to node 2 by edge `c`, and so on.

Control structure testing

Control structure testing is used to enhance the coverage area by testing various control structures (which include *logical structures* and *loops*) present in the program. Note that basis path testing is used as one of the techniques for control structure testing. Various types of testing performed under control structure testing are *condition testing*, *data-flow testing*, and *loop testing*.

Condition testing

In condition testing, the test cases are derived to determine whether the logical conditions and decision statements are free from errors. The errors presenting logical conditions

NOTES

can be incorrect Boolean operators, missing parenthesis in a Boolean expression, error in relational operators, arithmetic expressions, and so on.

The common types of logical conditions that are tested using condition testing are listed below.

- A relational expression, such as `E1 op E2`, where `E1` and `E2` are arithmetic expressions and `op` is an operator.
- A simple condition, such as any relational expression proceeded by a `NOT` (~) operator. For example, (~ `E1`), where `E1` is an arithmetic expression.
- A compound condition, which is composed of two or more simple conditions, Boolean operators, and parenthesis. For example, (`E1 & E2`) `|` (`E2 & E3`), where `E1`, `E2`, and `E3` are arithmetic expressions and `&` and `|` represent `AND` and `OR` operators.
- A Boolean expression consisting of operands and a Boolean operator, such as `AND`, `OR`, or `NOT`. For example, `A | B` is a Boolean expression, where `A` and `B` are operands and `|` represents `OR` operator.

Condition testing is performed using different strategies, namely, *branch testing*, *domain testing*, and *branch and relational operator testing*. **Branch testing** executes each branch (like 'if' statement) present in the module of a program at least once to detect all the errors present in the branch. **Domain testing** tests relational expressions present in a program. For this, domain testing executes all statements of the program that contain relational expressions. **Branch and relational operator testing** tests the branches present in the module of a program using condition constraints. For example,

```
if a >10
then
print big
```

In this case, branch and relational operator testing verifies that when the above code is executed, it produces the output 'big' only if the value of variable `a` is greater than 10.

Data-flow testing

In data-flow testing, test cases are derived to determine the validity of variables definitions and their uses in the program. This testing ensures that all variables are used properly in a program. To specify test cases, data-flow-based testing uses information such as location at which the variables are defined and used in the program.

For performing data-flow testing, a **definition-use graph** is built by associating the program variables with nodes and edges of the control flow graph. Once these variables are attached, test cases can easily determine which variable is used in which part of a program and how data is flowing in the program. Thus, data-flow of a program can be tested easily using specified test cases.

NOTES

Loop testing

Loop testing is used to check the validity of loops present in the program modules. Generally, there exist four types of loops (see Figure 10.24), which are discussed below.

- **Simple loop:** Refers to a loop that has no other loops in it. Consider a simple loop of size `n`. Size `n` of the loop indicates that the loop can be traversed `n` times, that is, `n` number of passes are made through the loop. The steps followed for testing simple loops are given below.
  1. Skip the entire loop.
  2. Traverse the loop only once.
  3. Traverse the loop two times.
  4. Make `m` number of passes through the loop, where `m` < `n`.
  5. Traverse the loop `n`-1, `n`, `n` + 1 times.
- **Nested Loops:** Loops within loops are known as nested loops. The number of tests required for testing nested loops depends on the level of nesting. More is the level of nesting; more will be the number of tests required. The steps followed for testing nested loops are listed below.
  1. Start with the inner loop and set values of all the outer loops to minimum.
  2. Test the inner loop using the steps followed for testing simple loops while keeping the iteration parameters of the outer loops at their minimum values. Add other tests for values that are either out-of-range or are eliminated.
  3. Move outwards, conducting tests for the next loop while holding other nested loops to 'typical' values and the iteration parameters of the outer loops at their minimum values.
  4. Continue testing until all loops are tested.
- **Concatenated loops:** The loops containing several loops that may be dependent or independent. In case the loops are dependent on each other, the steps in nested loops are followed. On the other hand, if the loops are independent of each other, the steps in simple loops are followed.
- **Unstructured Loops:** Such loops are difficult to test; therefore, they should be redesigned so that the use of structured programming constructs can be reflected.

NOTES

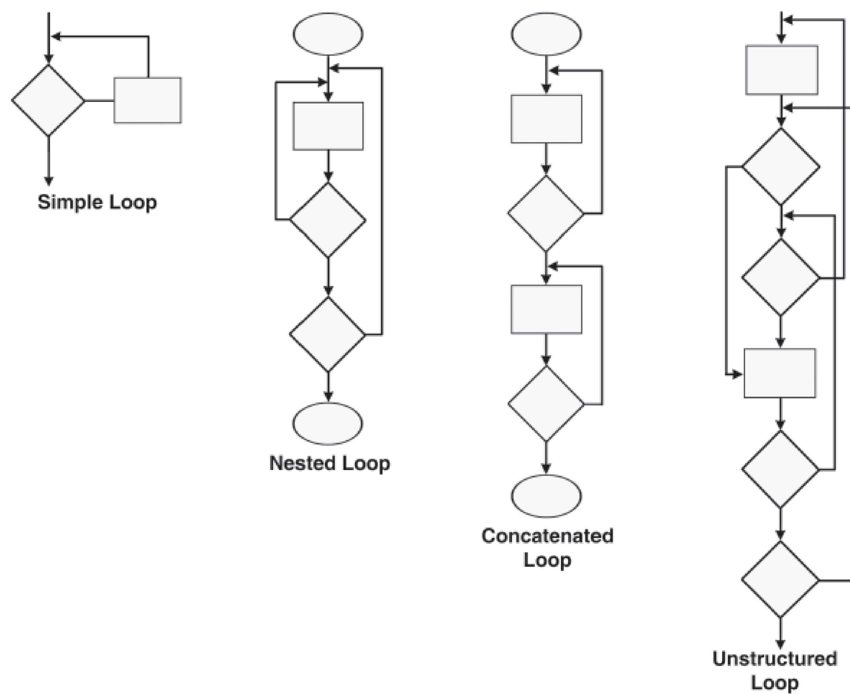


Fig. 10.24 *Types of Loops*

Mutation testing

Mutation testing is a white box method where errors are 'purposely' inserted into a program (under test) to verify whether the existing test case is able to detect the error. In this testing, mutants of the program are created by making some changes in the original program. The objective is to check whether each mutant produces an output that is different from the output produced by the original program (see Figure 10.25).

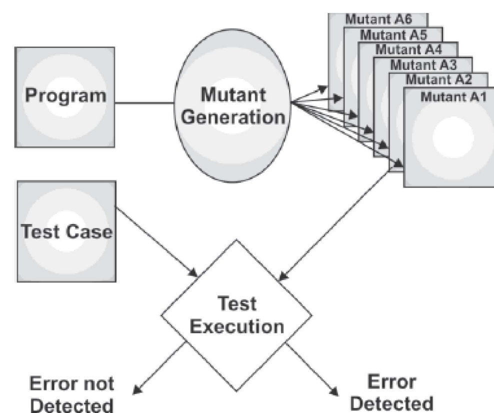


Fig. 10.25 *Mutation Testing*

In mutation testing, test cases that are able to 'kill' all the mutants should be developed. This is accomplished by testing mutants with the developed set of test cases. There can be two possible outcomes when the test cases test the program—

NOTES

either the test case detects the faults or fails to detect faults. If faults are detected, then necessary measures are taken to correct them.

When no faults are detected, it implies that either the program is absolutely correct or the test case is inefficient to detect the faults. Therefore, it can be concluded that mutation testing is conducted to determine the effectiveness of a test case. That is, if a test case is able to detect these 'small' faults (minor changes) in a program, then it is likely that the same test case will be equally effective in finding real faults.

To perform mutation testing, a number of steps are followed, which are listed below.

1. Create mutants of a program.
2. Check both program and its mutants using test cases.
3. Find the mutants that are different from the main program. A mutant is said to be different from the main program if it produces an output, which is different from the output produced by the main program.
4. Find mutants that are equivalent to the program. A mutant is said to be equivalent to the main program if it produces the same output as that of the main program.
5. Compute the mutation score using the formula given below.
   `M = D /(N-E)`
   Where `M` = Mutation score
   `N` = Total number of mutants of the program
   `D` = Number of mutants different from the main program
   `E` = Total number of mutants that are equivalent to the main program.
6. Repeat steps 1 to 5 till the mutation score is '1'.

However, mutation testing is very expensive to run on large programs. Thus, certain tools are used to run mutation tests on large programs. For example, 'Jester' is used to run mutation tests on Java code. This tool targets the specific areas of the program code, such as changing constants and Boolean values.

10.3.2 Black Box Testing

Black box (or **functional**) testing checks the functional requirements and examines the input and output data of these requirements (see Figure 10.26). When black box testing is performed, only the sets of 'legal' input and corresponding outputs should be known to the tester and not the internal logic of the program to produce that output. Hence to determine the functionality, the outputs produced for the given sets of input are observed.

NOTES

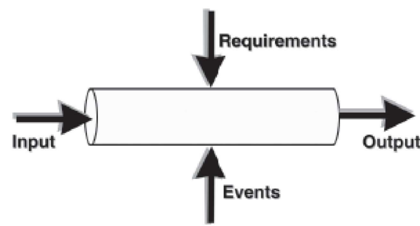


Fig. 10.26 Black Box Testing

The black box testing is used to find the following errors (see Figure 10.27).

- Interface errors, such as functions, which are unable to send or receive data to/from other software.
- Incorrect functions that lead to undesired output when executed.
- Missing functions and erroneous data structures.
- Erroneous databases, which lead to incorrect outputs when software uses the data present in these databases for processing.
- Incorrect conditions due to which the functions produce incorrect outputs when they are executed.
- Termination errors, such as certain conditions due to which a function enters a loop that forces it to execute indefinitely.

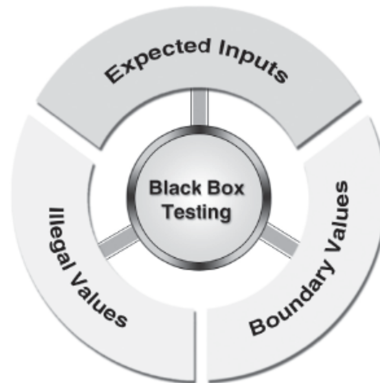


Fig. 10.27 Types of Error Detection in Black Box Testing

In this testing, tester derives various test cases to exercise the functional requirements of the software without considering implementation details of the code. Then, the software is run for the specified sets of input and the outputs produced for each input set is compared against the specifications to conform the correctness. If they are as specified by the user, then the software is considered to be correct else the software is tested for the presence of errors in it. The advantages and disadvantages associated with black box testing are listed in Table 10.8.

Table 10.8 Advantages and Disadvantages of Black box Testing.

NOTES

Advantages	Disadvantages
• Tester requires no knowledge of implementation and programming language used.	• Only a small number of possible inputs can be tested as testing every possible input consumes a lot of time.
• Reveals any ambiguities and inconsistencies in the functional specifications.	• There can be unnecessary repetition of test inputs if the tester is not informed about the test cases that the software developer has already tried.
• Efficient when used on larger systems.	• Leaves many program paths untested.
• A non-technical person can also perform black box testing.	• Cannot be directed towards specific segments of code, hence is more error prone.

Various methods used in black box testing are equivalence class partitioning, boundary value analysis, and cause-effect graphing (see Figure 10.28). In **equivalence class partitioning**, the test inputs are classified into equivalence classes such that one input checks (validates) all the input values in that class. In **boundary value analysis**, the boundary values of the equivalence classes are considered and tested. In **cause effect graphing,** cause-effect graphs are used to design test cases, which provides all the possible combinations of inputs to the program.

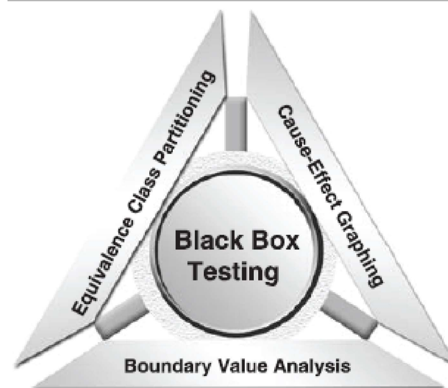


Fig. 10.28 *Types of Black Box Testing*

Equivalence class partitioning

This method tests the validity of outputs by dividing the input domain into different classes of data (known as equivalence classes) using which test cases can be easily generated (see Figure 10.29). Test cases are designed with the purpose of covering each partition at least once. If a test case is able to detect all the errors in the specified partition, then the test case is said to be an ideal test case.

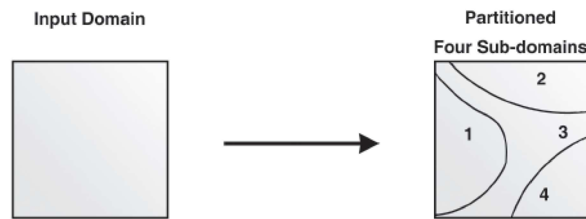


Fig. 10.29 Input Domain and Equivalence Classes

NOTES

An equivalence class depicts valid or invalid states for the input condition. An input condition can be either a specific numeric value, a range of values, a Boolean condition, or a set of values. The general guidelines that are followed for generating the equivalence classes are listed in Table 10.9.

Table 10.9 Guidelines for Generating Equivalence Classes

Input Condition	Number of Equivalence Classes	Description
Boolean	Two	One valid and one invalid
Specific numeric value	Three	One valid and two invalid
Range	Three	One valid and two invalid
Member of a set	Two	One valid and one invalid

To understand equivalence class partitioning properly, let us consider an example. This example is explained in the series of steps listed below.

1. Suppose that a program `P` takes an integer `X` as input.
2. Now, either `X` < 0 or `X` ≥ 0.
3. In case `X` < 0, the program is required to perform task `T1`; otherwise, task `T2` is performed.
4. The input domain is as large as `X` and it can assume a large number of values. Therefore the input domain (`P`) is partitioned into two equivalence classes and all test inputs in the `X` < 0 and `X` ≥ 0 equivalence classes are considered to be equivalent.
5. Now, as shown in Figure 10.30 independent test cases are developed for `X` < 0 and `X` ≥ 0.

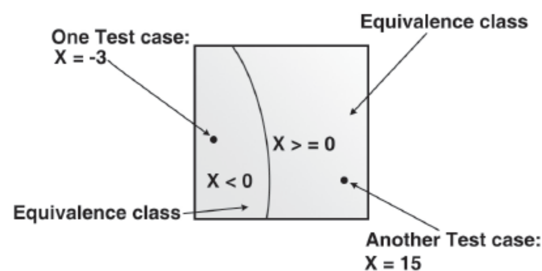


Fig. 10.30 Test Case and Equivalence Class

NOTES

Boundary value analysis

In boundary value analysis (BVA), test cases are derived on the basis of values that lie on an edge of the equivalence partitions. These values can be input or output values either at the edge or within the permissible range from the edge of an equivalence partition.

BVA is used since it has been observed that most errors occur at the boundary of input domain rather than at the middle of the input domain. Note that boundary value analysis complements the equivalence partitioning method. The only difference is that in BVA, test cases are derived for both input domain and output domain while in equivalence partitioning test cases are derived only for input domain.

Generally, the test cases are developed in boundary value analysis using certain guidelines, which are listed below.

- If an input condition specifies a range of values, test cases should be developed on the basis of both the values at the boundaries and the values that are just above and below the boundary values. For example, for the range $-0.5 \leq X \leq 0.5$, the input values for a test case can be '-0.4', -'0.5', '0.5', '0.6'.
- If an input condition specifies a number of values, test cases should be designed to exercise the minimum and maximum numbers as well as values just above and below these numbers.
- If an input consists of certain data structures (like arrays), then the test case should be able to execute all the values present at the boundaries of the data structures such as the maximum and minimum value of an array.

Cause-effect graphing

Equivalence partitioning and boundary value analysis tests each input given to a program independently. It means none of these consider the case of combinations of inputs, which may produce situations that need to be tested. This drawback is avoided in cause-effect graphing where combinations of inputs are used instead of individual inputs. In this technique, the causes (input conditions) and effects (output conditions) of the system are identified and a graph is created with each condition as the node of the graph. This graph is called **cause-effect graph**. This graph is then used to derive test cases. To use the cause-effect graphing method, a number of steps are followed, which are listed below.

1. List the cause (input conditions) and effects (outputs) of the program.
2. Create a cause-effect graph.
3. Convert the graph into a decision table.
4. Generate test cases from the decision table rules.

In order to generate test cases, all causes and effects are allocated unique numbers, which are used to identify them. After allocating numbers, the cause due

NOTES

to which a particular effect occurred is determined. Next, the combinations of various conditions that make the effect 'true' are recognized. A condition has two states, 'true' and 'false'. A condition is 'true' if it causes the effect to occur; otherwise, it is 'false'. The conditions are combined using Boolean operators such as 'AND' (&), 'OR' (|), and 'NOT' (~). Finally, a test case is generated for all possible combinations of conditions.

Various symbols are used in the cause-effect graph (see Figure 10.31). The figure depicts various *logical* associations among causes c_i and effects e_i. The dashed notation on the right side in the figure indicates various *constraint* associations that can be applied to either causes or effects.

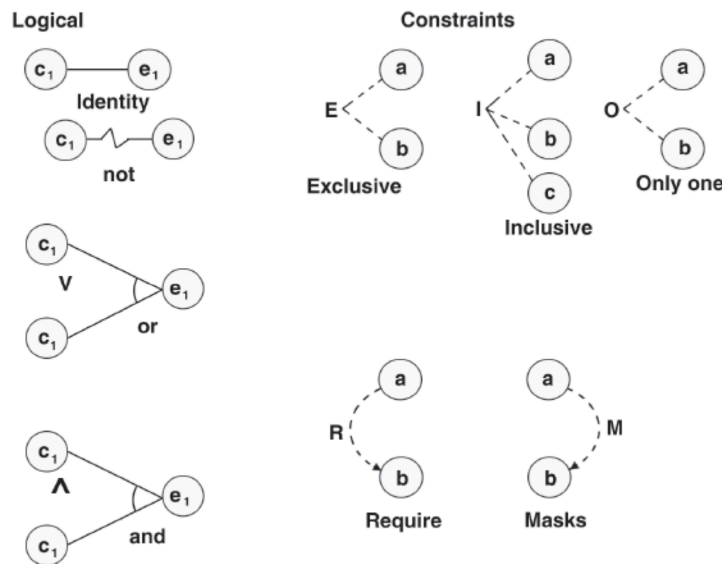


Fig. 10.31 *Logical and Constraints Associations*

To understand cause-effect graphing properly, let us consider an example. Suppose a triangle is drawn with inputs `x`, `y`, and `z`. The values of these inputs are given between 0 and 100. Using these inputs, three outputs are produced, namely, isosceles triangle, equilateral triangle or no triangle is made (if values of `x`, `y`, `z` are less than 60º).

1. Using the steps of cause-effect graphing, initially the causes and effects of the problem are recognized, which are listed in Table 10.10.

Table 10.10 *Causes and Effects*

Causes	Effects
C1: side x is less than the sum of sides y and z. C2: sides x, y, z are equal. C3: side x is equal to side y. C4: side y is equal to side z. C5: side x is equal to side z.	E1: no triangle is formed. E2: equilateral triangle is formed. E3: isosceles triangle is formed.

NOTES

2. The cause-effect graph is generated as shown in Figure 10.32.

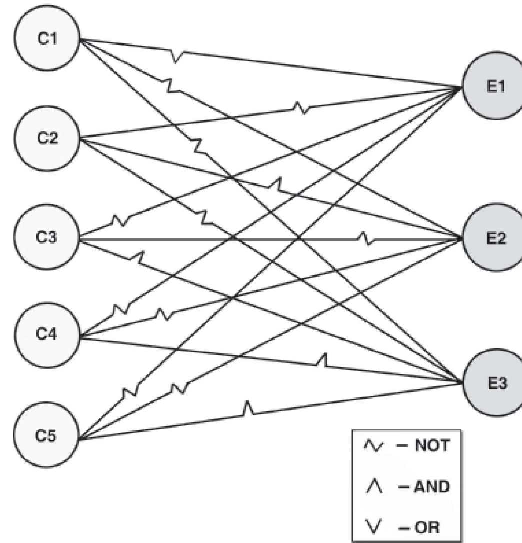


Fig. 10.32 *Cause-Effect Graph*

3. A decision table (a table that shows a set of conditions and the actions resulting from them) is drawn as shown in Table 10.11.

Table 10.11 *Decision Table*

Conditions					
`C1: x < y + z`	0	X	X	X	X
`C2: x = y = z`	X	1	X	X	X
`C3: x = y`	X	X	1	X	X
`C4: y = z`	X	X	X	1	X
`C5: x = z`	X	X	X	X	1
Actions					
`E1`: not a triangle	1				
`E2`: equilateral triangle		1			
`E3`: isosceles triangle			1	1	1

4. Each combination of conditions for an effect in Table 10.11 is a test case.

Differences between Black Box and White Box Testing

Although white box testing and black box testing are used together for testing many programs, there are several considerations that make them different from each other. Black box testing detects errors of *omission*, which are errors occurring due to non-accomplishment of user requirements. On the other hand, white box testing detects errors of *commission* which are errors occurring due to non-implementation of some part of software code. The other differences between white box testing and black box testing are listed in Table 10.12.

Table 10.12 Differences between White Box and Black Box Testing

Basis	White Box Testing	Black Box Testing
Purpose	▪ To test the internal structure of software. ▪ Test the software but does not ensure the complete implementation of all the specifications mentioned in user requirements. ▪ Addresses flow and control structure of a program.	▪ To test the functionality of software. ▪ Concerned with testing the specifications and does not ensure that all the components of software that are implemented are tested. ▪ Addresses validity, behavior and performance of software
Stage	▪ Performed in the early stages of testing.	▪ Performed in the later stages of testing.
Requirement	▪ Knowledge of the internal structure of a program is required for generating test case.	▪ No knowledge of the internal structure of a program is required to generate test case.
Test Cases	▪ Test cases are generated on the basis of the internal structure or code of the module to be tested.	▪ Internal structure of modules or programs is not considered for selecting test cases.
Example	▪ The inner software present inside the calculator (which is known by the developer only) is checked by giving inputs to the code.	▪ In this testing, it is checked whether the calculator is working properly by giving inputs by pressing the buttons in the calculator.

NOTES

10.4 DEBUGGING

On successful culmination of software testing, debugging is performed. **Debugging** is defined as a process of analyzing and removing the error. It is considered necessary in most of the newly developed software or hardware and in commercial products/ personal application programs. For complex products, debugging is done at all the levels of the testing.

Debugging is considered to be a complex and time-consuming process since it attempts to remove errors at all the levels of testing. To perform debugging, debugger (debugging tool) is used to reproduce the conditions in which failure occurred, examine the program state, and locate the cause. With the help of debugger, programmers trace the program execution step by step (evaluating the value of variables) and halt the execution wherever required to reset the program variables. Note that some programming language packages include a debugger for checking the code for errors while it is being written.

Guidelines for Debugging

NOTES

Some guidelines that are followed while performing debugging are discussed here.

- Debugging is the process of solving a problem. Hence, individuals involved in debugging should understand all the causes of an error before starting with debugging.
- No experimentation should be done while performing debugging. The experimental changes instead of removing errors often increase the problem by adding new errors in it.
- When there is an error in one segment of a program, there is a high possibility that some other errors also exist in the program. Hence, if an error is found in one segment of a program, rest of the program should be properly examined for errors.
- It should be ensured that the new code added in a program to fix errors is correct and is not introducing any new error in it. Thus, to verify the correctness of a new code and to ensure that no new errors are introduced, regression testing should be performed.

10.4.1 The Debugging Process

During debugging, errors are encountered that range from less damaging (like input of an incorrect function) to catastrophic (like system failure, which lead to economic or physical damage). Various levels of errors and their damaging effects are shown in Figure 10.33. Note that with the increase in number of errors, the amount of effort to find their causes also increases.

Once errors are identified in a software system, to debug the problem, a number of steps are followed, which are listed below.

1. ***Defect confirmation/identification***: A problem is identified in a system and a defect report is created. A software engineer maintains and analyzes this error report and finds solutions to the following questions.
   - Does a defect exist in the system?
   - Can the defect be reproduced?
   - What is the expected/desired behavior of the system?
   - What is the actual behavior?

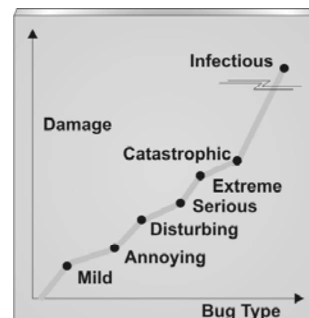


Fig. 10.33 *Levels of Error and its Damaging Effect*

2. ***Defect analysis***: If the defect is genuine, the next step is to understand the root cause of the problem. Generally, engineers debug by starting a debugging tool (debugger) and they try to understand the root cause of the problem by following a step-by-step execution of the program.
3. ***Defect resolution***: Once the root cause of a problem is identified, the error can be resolved by making an appropriate change to the system by fixing the problem.

When the debugging process ends, the software is retested (as shown in Figure 10.34) to ensure that no errors are left undetected. Moreover, it checks that no new errors are introduced in the software while making some changes to it during the debugging process.

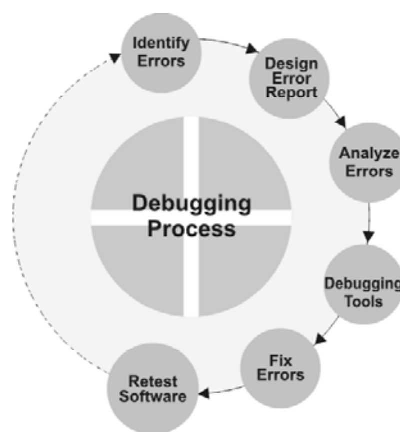


Fig. 10.34 *Debugging Process*

Debugging Strategies

As debugging is a difficult and time-consuming task, it is essential to develop a proper debugging strategy. This strategy helps in performing the process of debugging easily and efficiently. The commonly-used debugging strategies are *debugging by brute force*, *induction strategy*, *deduction strategy*, *backtracking strategy*, and *debugging by testing* (see Figure 10.35).

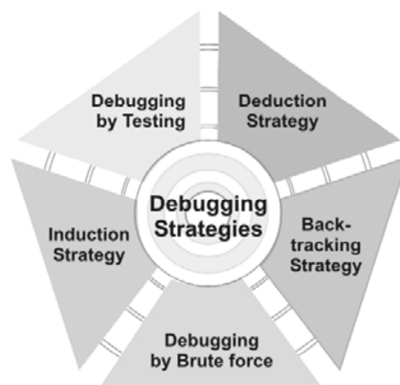


Fig. 10.35 *Debugging Strategies*

NOTES

Debugging by Brute Force

Brute force method of debugging is the most commonly used but least efficient method. It is generally used when all other available methods fail. Here, debugging is done by taking memory (or storage) dumps. Actually, the program is loaded with the output statements that produce a large amount of information including the intermediate values. Analyzing this information may help to identify the errors cause. However, using a memory dump for finding errors requires analyzing huge amount of information or irrelevant data leading to waste of time and effort.

10.4.2 Induction Strategy

This strategy is a 'disciplined thought process' where errors can be debugged by moving outwards from the *particulars to the whole.* This strategy assumes that once the symptoms of the errors are identified, and the relationships between them are established, the errors can be easily detected by just looking at the symptoms and the relationships. To perform induction strategy, a number of steps are followed, which are listed below.

1. ***Locating relevant data***: All the information about a program is collected to identify the functions, which are executed correctly and incorrectly.
2. ***Organizing data***: The collected data is organized according to importance. The data can consist of possible symptoms of errors, their location in the program, the time at which the symptoms occur during the execution of the program and the effect of these symptoms on the program.
3. ***Devising hypothesis***: The relationships among the symptoms are studied and a hypothesis is devised that provides the hints about the possible causes of errors.
4. ***Proving hypothesis***: In the final step, the hypothesis needs to be proved. It is done by comparing the data in the hypothesis with the original data to ensure that the hypothesis explains the existence of hints completely. In case, the hypothesis is unable to explain the existence of hints, it is either incomplete or contains multiple errors in it.

Deduction Strategy

In this strategy, first step is to identify all the possible causes and then using the data each cause is analyzed and eliminated if it is found invalid. Note that as in induction strategy, deduction strategy is also based on some assumptions. To use this strategy following steps are followed.

1. ***Identifying the possible causes or hypotheses***: A list of all the possible causes of errors is formed. Using this list, the available data can be easily structured and analyzed.
2. ***Eliminating possible causes using the data***: The list is examined to recognize the most probable cause of errors and the rest of the causes are deleted.

3. ***Refining the hypothesis***: By analyzing the possible causes one by one and looking for contradiction leads to elimination of invalid causes. This results in a refined hypothesis containing few specific possible causes.
4. ***Proving the hypothesis***: This step is similar to the fourth step in induction method.

Backtracking Strategy

This method is effectively used for locating errors in small programs. According to this strategy, when an error has occurred, one needs to start tracing the program backward one step at a time evaluating the values of all variables until the cause of error is found. This strategy is useful but in a large program with many thousands lines of code, the number of backward paths increases and becomes unmanageably large.

Debugging by Testing

This debugging method can be used in conjunction with debugging by induction and debugging by deduction methods. Additional test cases are designed that help in obtaining information to devise and prove a hypothesis in induction method and to eliminate the invalid causes and refine the hypothesis in deduction method. Note that the test cases used in debugging are different from the test cases used in testing process. Here, the test cases are specifically designed to explore the internal program state.

Check Your Progress

4. What is beta testing.
5. Write a note on mutation testing.
6. What is debugging?

10.5 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Software testing determines the correctness, completeness and quality of software being developed.
2. The ease with which a program is tested is known as testability.
3. A test plan describes how testing would be accomplished. It is a document that specifies the purpose, scope, and method of software testing.
4. Beta testing assesses the performance of the software at user’s site. This testing is ‘live’ testing and is conducted in an environment, which is not controlled by the developer.

NOTES

5. Mutation testing is a white box method where errors are 'purposely' inserted into a program (under test) to verify whether the existing test case is able to detect the error.
6. Debugging is defined as a process of analyzing and removing the error. It is considered necessary in most of the newly developed software or hardware and in commercial products/personal application programs.

10.6 SUMMARY

- Software testing is performed either manually or by using automated tools to make sure that the software is functioning in accordance with the user requirements.
- The ease with which a program is tested is known as testability. Testability should always be considered while designing and implementing a software system so that the errors in the system can be detected with minimum effort.
- A testing strategy is used to identify the levels of testing which are to be applied along with the methods, techniques, and tools to be used during testing.
- Unit testing is performed to test the individual units of software. Since the software comprises various units/modules, detecting errors in these units is simple and consumes less time, as they are small in size.
- The integration testing is aimed at ensuring that all the modules work properly as per the user requirements when they are put together.
- Smoke testing is defined as an approach of integration testing in which a subset of test cases designed to check the main functionality of software are used to test whether the vital functions of the software work correctly.
- Alpha testing is considered as a form of internal acceptance testing in which the users test the software at the developer's site.
- Beta testing assesses the performance of the software at user's site. This testing is 'live' testing and is conducted in an environment, which is not controlled by the developer.
- Stress testing is designed to determine the behavior of the software under abnormal situations.
- Performance testing is designed to determine the performance of the software at the run-time in the context of the entire computer-based system.
- White box testing, also known as structural testing or glass box testing, is performed to check the internal structure of a program.
- Graph matrix is used to develop software tool that in turn helps in carrying out basis path testing. It is defined as a data structure used to represent the flow graph of a program in a tabular form.

10.7 KEY WORDS

NOTES

- **Validation Testing:** The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements. Validation Testing ensures that the product actually meets the client's needs.
- **Smoke Testing:** It is defined as an approach of integration testing in which a subset of test cases designed to check the main functionality of software are used to test whether the vital functions of the software work correctly.
- **Regression Testing:** It 're-tests' the software or part of it to ensure that the components, features, and functions, which were previously working properly, do not fail as a result of the error correction process and integration of modules.

10.8 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Define software testing.
2. What are the characteristics of software test?
3. What is test plan?
4. What are software testing strategies?
5. What are the various levels of testing?
6. Differentiate between white box and black box testing.

Long Answer Questions

1. What are the guidelines of software testing? Explain.
2. Explain the various types of software testing strategies.
3. How will you develop a software testing strategy?
4. What do you understand by integration and regression testing?
5. Explain the different types of system testing.
6. Explain white box and black box testing.

10.9 FURTHER READINGS

Pressman, Roger S. 1997. *Software Engineering, a Practitioner's Approach*. New Delhi: Tata McGraw-Hill.

NOTES

Jalote, Pankaj. 1991. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

Schach, Stephen R. 2005. *Object Oriented and Classical Software Engineering*. New Delhi: Tata McGraw-Hill.

Somerville, Ian. 2001. *Software Engineering*. New Delhi: Pearson Education.

Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandriolli . 1991. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hill of India.

Jawadekar, Waman S. 2004. *Software Engineering: Principles and Practice*. New Delhi: Tata McGraw-Hill.

UNIT 11 PRODUCT METRICS

NOTES

Structure

11.0 INTRODUCTION

To achieve an accurate schedule and cost estimate, better quality products, and higher productivity, an effective software management is required, which in turn can be attained through the use of software metrics. A metric is a derived unit of measurement that cannot be directly observed, but is created by combining or relating two or more measures. Product metrics is the measurement of work product produced during different phases of software development.

Various studies suggest that careful implementation and application of software metrics helps in achieving better management results, both in the short run (a given project) and the long run (improving productivity of future projects). Effective metrics not only describe the models that are capable of predicting process or product parameters, but also facilitates the development of these models. An ideal metrics should be simple and precisely defined, easily obtainable, valid, and robust.

11.1 OBJECTIVES

After going through this unit, you will be able to:

- Understand and design software metrics

- Describe process, product and project metrics
- Discuss the issues in software metrics

NOTES

11.2 SOFTWARE MEASUREMENT

To assess the quality of the engineered product or system and to better understand the models that are created, some measures are used. These measures are collected throughout the software development life cycle with an intention to improve the software process on a continuous basis. Measurement helps in estimation, quality control, productivity assessment and project control throughout a software project. Also, measurement is used by software engineers to gain insight into the design and development of the work products. In addition, measurement assists in strategic decision-making as a project proceeds.

Software measurements are of two categories, namely, *direct measures* and *indirect measures*. **Direct measures** include software processes like cost and effort applied and products like lines of code produced, execution speed, and other defects that have been reported. **Indirect measures** include products like functionality, quality, complexity, reliability, maintainability, and many more.

Generally, software measurement is considered as a management tool which if conducted in an effective manner, helps the project manager and the entire software team to take decisions that lead to successful completion of the project. Measurement process is characterized by a set of five activities, which are listed below.

- **Formulation:** This performs measurement and develops appropriate metric for software under consideration.
- **Collection:** This collects data to derive the formulated metrics.
- **Analysis:** This calculates metrics and the use of mathematical tools.
- **Interpretation:** This analyzes the metrics to attain insight into the quality of representation.
- **Feedback:** This communicates recommendation derived from product metrics to the software team.

Note that collection and analysis activities drive the measurement process. In order to perform these activities effectively, it is recommended to automate data collection and analysis, establish guidelines and recommendations for each metric, and use statistical techniques to interrelate external quality features and internal product attributes.

11.3 SOFTWARE METRICS

Once measures are collected they are converted into metrics for use. **IEEE** defines metric as '*a quantitative measure of the degree to which a system, component,*

or process possesses a given attribute.' The goal of software metrics is to identify and control essential parameters that affect software development. Other objectives of using software metrics are listed below.

NOTES

- Measuring the size of the software quantitatively.
- Assessing the level of complexity involved.
- Assessing the strength of the module by measuring coupling.
- Assessing the testing techniques.
- Specifying when to stop testing.
- Determining the date of release of the software.
- Estimating cost of resources and project schedule.

Software metrics help project managers to gain an insight into the efficiency of the software process, project, and product. This is possible by collecting quality and productivity data and then analyzing and comparing these data with past averages in order to know whether quality improvements have occurred. Also, when metrics are applied in a consistent manner, it helps in project planning and project management activity. For example, schedule-based resource allocation can be effectively enhanced with the help of metrics.

Difference in Measures, Metrics, and Indicators

Metrics is often used interchangeably with measure and measurement. However, it is important to note the differences between them. **Measure** can be defined as quantitative indication of amount, dimension, capacity, or size of product and process attributes. **Measurement** can be defined as the process of determining the measure. **Metrics** can be defined as quantitative measures that allow software engineers to identify the efficiency and improve the quality of software process, project, and product.

To understand the difference, let us consider an example. A measure is established when a number of errors is (single data point) detected in a software component. Measurement is the process of collecting one or more data points. In other words, measurement is established when many components are reviewed and tested individually to collect the measure of a number of errors in all these components. Metrics are associated with individual measure in some manner. That is, metrics are related to detection of errors found per review or the average number of errors found per unit test.

Once measures and metrics have been developed, **indicators** are obtained. These indicators provide a detailed insight into the software process, software project, or intermediate product. Indicators also enable software engineers or project managers to adjust software processes and improve software products, if required. For example, measurement dashboards or key indicators are used to monitor progress and initiate change. Arranged together, indicators provide snapshots of the system's performance.

NOTES

Measured Data

Before data is collected and used, it is necessary to know the type of data involved in the software metrics. Table 11.1 lists different types of data, which are identified in metrics along with their description and the possible operations that can be performed on them.

Table 11.1 *Type of Data Measured*

Type of Data	*Possible Operations*	*Description of Data*
Nominal	=, ≠	Categories
Ordinal	<, >	Ranking
Interval	+, -	Differences
Ratio	/	Absolute zero

- **Nominal data:** Data in the program can be measured by placing it under a category. This category of program can be a database program, application program, or an operating system program. For such data, operation of arithmetic type and ranking of values in any order (increasing or decreasing) is not possible. The only operation that can be performed is to determine whether program 'X' is the same as program 'Y'.
- **Ordinal data:** Data can be ranked according to the data values. For example, experience in application domain can be rated as very low, low, medium, or high. Thus, experience can easily be ranked according to its rating.
- **Interval data:** Data values can be ranked and substantial differences between them can also be shown. For example, a program with complexity level 8 is said to be 4 units more complex than a program with complexity level 4.
- **Ratio data:** Data values are associated with a ratio scale, which possesses an absolute zero and allows meaningful ratios to be calculated. For example, program lines expressed in lines of code.

It is desirable to know the measurement scale for metrics. For example, if metrics values are used to represent a model for a software process, then metrics associated with the ratio scale may be preferred.

Guidelines for Software Metrics

Although many software metrics have been proposed over a period of time, ideal software metric is the one which is easy to understand, effective, and efficient. In order to develop ideal metrics, software metrics should be validated and characterized effectively. For this, it is important to develop metrics using some specific guidelines, which are listed below.

NOTES

- **Simple and computable:** Derivation of software metrics should be easy to learn and should involve average amount of time and effort.
- **Consistent and objective:** Unambiguous results should be delivered by software metrics.
- **Consistent in the use of units and dimensions:** Mathematical computation of the metrics should involve use of dimensions and units in a consistent manner.
- **Programming language independent:** Metrics should be developed on the basis of the analysis model, design model, or program's structure.
- **High quality:** Effective software metrics should lead to a high-quality software product.
- **Easy to calibrate:** Metrics should be easy to adapt according to project requirements.
- **Easy to obtain:** Metrics should be developed at a reasonable cost.
- **Validation:** Metrics should be validated before being used for making any decisions.
- **Robust:** Metrics should be relatively insensitive to small changes in process, project, or product.
- **Value:** Value of metrics should increase or decrease with the value of the software characteristics they represent. For this, the value of metrics should be within a meaningful range. For example, metrics can be in a range of 0 to 5.

11.4 DESIGNING SOFTWARE METRICS

An effective software metrics helps software engineers to identify shortcomings in the software development life cycle so that the software can be developed as per the user requirements, within estimated schedule and cost, with required quality level, and so on. To develop effective software metrics, the following steps are used.

1. **Definitions:** To develop an effective metric, it is necessary to have a clear and concise definition of entities and their attributes that are to be measured. Terms like defect, size, quality, maintainability, user-friendly, and so on should be well defined so that no issues relating to ambiguity occur.
2. **Define a model:** A model for the metrics is derived. This model is helpful in defining how metrics are calculated. The model should be easy to modify according to the future requirements. While defining a model, the following questions should be addressed.
   - Does the model provide more information than is available?

NOTES

- Is the information practical?
- Does it provide the desired information?

3. **Establish counting criteria:** The model is broken down into its lowest-level metric entities and the counting criteria (which are used to measure each entity) are defined. This specifies the method for the measurement of each metric primitive. For example, to estimate the size of a software project, line of code (LOC) is a commonly used metric. Before measuring size in LOC, clear and specific counting criteria should be defined.
4. **Decide what is good:** Once it is decided what to measure and how to measure, it is necessary to determine whether action is needed. For example, if software is meeting the quality standards, no corrective action is necessary. However, if this is not true, then goals can be established to help the software conform to the quality standards laid down. Note that the goals should be reasonable, within the time frame, and based on supporting actions.
5. **Metrics reporting:** Once all the data for metric is collected, the data should be reported to the concerned person. This involves defining report format, data extraction and reporting cycle, reporting mechanisms, and so on.
6. **Additional qualifiers:** Additional metric qualifiers that are 'generic' in nature should be determined. In other words, metric that is valid for several additional extraction qualifiers should be determined.

The selection and development of software metrics is not complete until the effect of measurement and people on each other is known. The success of metrics in an organization depends on the attitudes of the people involved in collecting the data, calculating and reporting the metrics, and people involved in using these metrics. Also, metrics should focus on process, projects, and products and not on the individuals involved in this activity.

11.5 CLASSIFICATION OF SOFTWARE METRICS

As discussed earlier, measurement is done by metrics. Three parameters are measured: process measurement through *process metrics*, product measurement through *product metrics*, and project measurement through *project metrics*.

Process metrics assess the effectiveness and quality of software process, determine maturity of the process, effort required in the process, effectiveness of defect removal during development, and so on. **Product metrics** is the measurement of work product produced during different phases of software development. **Project metrics** illustrate the project characteristics and their execution.

11.6 PROCESS METRICS

NOTES

To improve any process, it is necessary to measure its specified attributes, develop a set of meaningful metrics based on these attributes, and then use these metrics to obtain indicators in order to derive a strategy for process improvement.

Using software process metrics, software engineers are able to assess the efficiency of the software process that is performed using the process as a framework. In Figure 11.1, process is placed at the centre of the triangle connecting three factors (product, people, and technology), which have an important influence on software quality and organization performance. The skill and motivation of the *people*, the complexity of the *product* and the level of technology used in the software development have an important influence on the quality and team performance. Also, in Figure 11.1, the process triangle exists within the circle of environmental conditions, which includes development environment, business conditions, and customer/user characteristics.

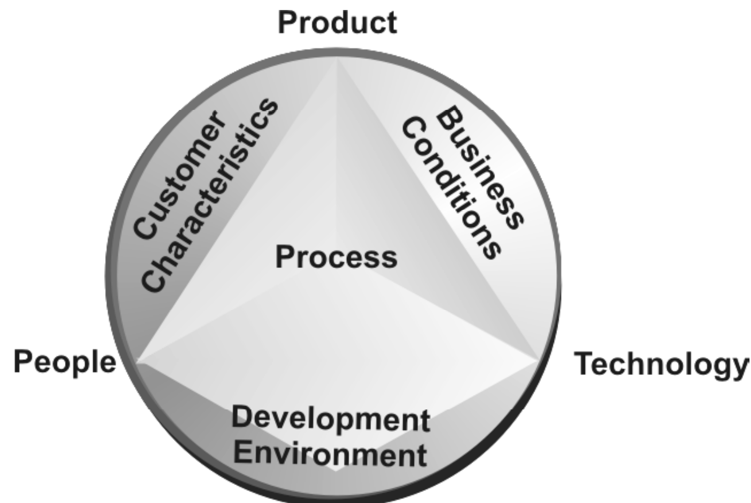


Fig. 11.1 *Process, Product, People, and Technology*

To measure the efficiency and effectiveness of the software process, a set of metrics is formulated based on the outcomes derived from the process. These outcomes are listed below.

- Number of errors found before the software release
- Defect detected and reported by the user after delivery of the software
- Time spent in fixing errors
- Work products delivered
- Human effort used
- Time expended
- Conformity to schedule

- Wait time
- Number of contract modifications
- Estimated cost compared to actual cost.

NOTES

Note that process metrics can also be derived using the characteristics of a particular software engineering activity. For example, an organization may measure the effort and time spent by considering the user interface design.

Types of Process Metrics

It is observed that process metrics are of two types, namely, *private* and *public*. **Private metrics** are private to the individual and serve as an indicator only for the specified individual(s). Defect rates by a software module and defect errors by an individual are examples of private process metrics. Note that some process metrics are public to all team members but private to the project. These include errors detected while performing formal technical reviews and defects reported about various functions included in the software.

Public metrics include information that was private to both individuals and teams. Project-level defect rates, effort and related data are collected, analyzed and assessed in order to obtain indicators that help in improving the organizational process performance.

Process Metrics Etiquette

Process metrics can provide substantial benefits as the organization works to improve its process maturity. However, these metrics can be misused and create problems for the organization. In order to avoid this misuse, some guidelines have been defined, which can be used both by managers and software engineers. These guidelines are listed below.

- Rational thinking and organizational sensitivity should be considered while analyzing metrics data.
- Feedback should be provided on a regular basis to the individuals or teams involved in collecting measures and metrics.
- Metrics should not appraise or threaten individuals.
- Since metrics are used to indicate a need for process improvement, any metric indicating this problem should not be considered harmful.
- Use of single metrics should be avoided.

As an organization becomes familiar with process metrics, the derivation of simple indicators leads to a stringent approach called **Statistical Software Process Improvement** (**SSPI**). SSPI uses software failure analysis to collect information about all *errors* (it is detected before delivery of the software) and *defects* (it is detected after software is delivered to the user) encountered during the development of a product or system.

11.7 PRODUCT METRICS

NOTES

In software development process, a working product is developed at the end of each successful phase. Each product can be measured at any stage of its development. Metrics are developed for these products so that they can indicate whether a product is developed according to the user requirements. If a product does not meet user requirements, then the necessary actions are taken in the respective phase.

Product metrics help software engineer to detect and correct potential problems before they result in catastrophic defects. In addition, product metrics assess the internal product attributes in order to know the efficiency of the following.

- Analysis, design, and code model
- Potency of test cases
- Overall quality of the software under development.

Various metrics formulated for products in the development process are listed below.

- **Metrics for analysis model:** These address various aspects of the analysis model such as system functionality, system size, and so on.
- **Metrics for design model:** These allow software engineers to assess the quality of design and include architectural design metrics, component-level design metrics, and so on.
- **Metrics for source code:** These assess source code complexity, maintainability, and other characteristics.
- **Metrics for testing:** These help to design efficient and effective test cases and also evaluate the effectiveness of testing.
- **Metrics for maintenance:** These assess the stability of the software product.

Metrics for the Analysis Model

There are only a few metrics that have been proposed for the analysis model. However, it is possible to use metrics for project estimation in the context of the analysis model. These metrics are used to examine the analysis model with the objective of predicting the size of the resultant system. Size acts as an indicator of increased coding, integration, and testing effort; sometimes it also acts as an indicator of complexity involved in the software design. Function point and lines of code are the commonly used methods for size estimation.

Function Point (FP) Metric

The function point metric, which was proposed by A.J Albrecht, is used to measure the functionality delivered by the system, estimate the effort, predict the number of

NOTES

errors, and estimate the number of components in the system. Function point is derived by using a relationship between the complexity of software and the information domain value. Information domain values used in function point include the number of external inputs, external outputs, external inquires, internal logical files, and the number of external interface files. Function point metric is discussed in detail in Chapter 10.

Lines of Code (LOC)

Lines of code (LOC) is one of the most widely used methods for size estimation. LOC can be defined as the number of delivered lines of code, excluding comments and blank lines. It is highly dependent on the programming language used as code writing varies from one programming language to another. For example, lines of code written (for a large program) in assembly language are more than lines of code written in C++.

From LOC, simple size-oriented metrics can be derived such as errors per KLOC (thousand lines of code), defects per KLOC, cost per KLOC, and so on. LOC has also been used to predict program complexity, development effort, programmer performance, and so on. For example, Hasltead proposed a number of metrics, which are used to calculate program length, program volume, program difficulty, and development effort. LOC is discussed in detail in Chapter 10.

Metrics for Specification Quality

To evaluate the quality of analysis model and requirements specification, a set of characteristics has been proposed. These characteristics include specificity, completeness, correctness, understandability, verifiability, internal and external consistency, achievability, concision, traceability, modifiability, precision, and reusability.

Most of the characteristics listed above are qualitative in nature. However, each of these characteristics can be represented by using one or more metrics. For example, if there are n_r requirements in a specification, then n_r can be calculated by the following equation.

$$n_r = n_f + n_{nf}$$

Where

n_f = number of functional requirements

n_{nf} = number of non-functional requirements.

In order to determine the **specificity** of requirements, a metric based on the consistency of the reviewer's understanding of each requirement has been proposed. This metric is represented by the following equation.

$$Q_1 = n_{ui}/n_r$$

Where

n_{ui} = number of requirements for which reviewers have same understanding

Q_1 = specificity.

Ambiguity of the specification depends on the value of Q. If the value of Q is close to 1 then the probability of having any ambiguity is less.

Completeness of the functional requirements can be calculated by the following equation.

$$Q_2 = n_u/[n_i*n_s]$$

Where

n_u = number of unique function requirements

n_i = number of inputs defined by the specification

n_s = number of specified state.

Q_2 in the above equation considers only functional requirements and ignores non-functional requirements. In order to consider non-functional requirements, it is necessary to consider the degree to which requirements have been validated. This can be represented by the following equation.

$$Q_3 = n_c/[n_c + n_{nv}]$$

Where

n_c = number of requirements validated as correct

n_{nv} = number of requirements, which are yet to be validated.

Metrics for Software Design

The success of a software project depends largely on the quality and effectiveness of the software design. Hence, it is important to develop software metrics from which meaningful indicators can be derived. With the help of these indicators, necessary steps are taken to design the software according to the user requirements. Various design metrics such as *architectural design metrics*, *component-level design metrics*, *user-interface design metrics*, and *metrics for object-oriented design* are used to indicate the complexity, quality, and so on of the software design.

Architectural Design Metrics

These metrics focus on the features of the program architecture with stress on architectural structure and effectiveness of components (or modules) within the architecture. In architectural design metrics, three software design complexity measures are defined, namely, *structural complexity*, *data complexity*, and *system complexity*.

In hierarchical architectures (call and return architecture), say module 'j', **structural complexity** is calculated by the following equation.

$$S(j) = f^2_{out}(j)$$

Where

$f_{out}(j)$ = fan-out of module 'j' [Here, fan-out means number of modules that are subordinating module j].

NOTES

Complexity in the internal interface for a module 'j' is indicated with the help of **data complexity,** which is calculated by the following equation.

$$D(j) = V(j)/[f_{out}(j) + 1]$$

Where

V(j) = number of input and output variables passed to and from module 'j'.

System complexity is the sum of structural complexity and data complexity and is calculated by the following equation.

$$C(j) = S(j) + D(j).$$

The complexity of a system increases with increase in structural complexity, data complexity, and system complexity, which in turn increases the integration and testing effort in the later stages.

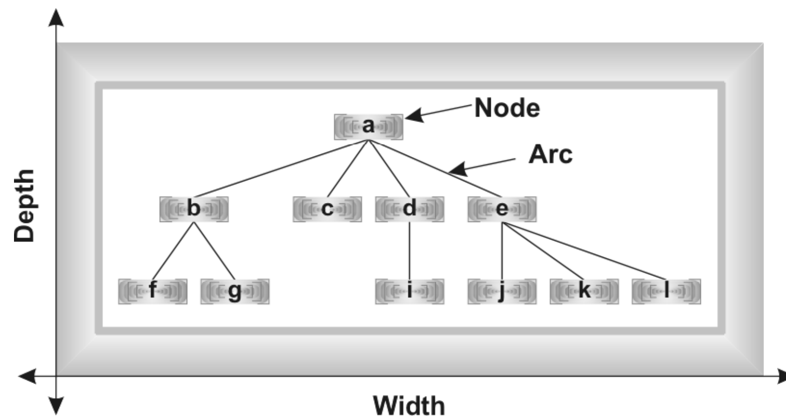


Fig. 11.2 *Morphology Metrics*

In addition, various other metrics like simple morphology metrics are also used. These metrics allow comparison of different program architecture using a set of straightforward dimensions. A metric can be developed by referring to call and return architecture as shown in Figure 11.2. This metric can be defined by the following equation.

$$Size = n + a$$

Where

n = number of nodes

a = number of arcs.

For example, in Figure 11.2, there are 11 nodes and 10 arcs. Here, Size can be calculated by the following equation.

$$Size = n + a = 11 + 10 = 21.$$

Depth is defined as the longest path from the top node (root) to the leaf node and width is defined as the maximum number of nodes at any one level. In Figure 11.2, the depth is 3 and the width is 6.

NOTES

Coupling of the architecture is indicated by arc-to-node ratio. This ratio also measures the connectivity density of the architecture and is calculated by the following equation.

$$r = a/n.$$

Quality of software design also plays an important role in determining the overall quality of the software. Many software quality indicators that are based on measurable design characteristics of a computer program have been proposed. One of them is **Design Structural Quality Index** (**DSQI**), which is derived from the information obtained from data and architectural design. To calculate DSQI, a number of steps are followed, which are listed below.

1. To calculate DSQI, the following values must be determined.
   - Number of components in program architecture (S_1)
   - Number of components whose correct function is determined by the source of input data (S_2)
   - Number of components whose correct function depends on previous processing (S_3)
   - Number of database items (S_4)
   - Number of different database items (S_5)
   - Number of database segments (S_6)
   - Number of components having single entry and exit (S_7).
2. Once all the values from S_1 to S_7 are known, some intermediate values are calculated, which are listed below.
   - **Program structure (D_1):** If discrete methods are used for developing architectural design then $D_1=1$, else $D_1 = 0$
   - **Module independence (D_2):** $D_2 = 1-(S_2/S_1)$
   - **Modules not dependent on prior processing (D_3):** $D_3 = 1-(S_3/S_1)$
   - **Database size (D_4):** $D_4 = 1-(S_5/S_4)$
   - **Database compartmentalization (D_5):** $D_5 = 1-(S_6/S_4)$
   - **Module entrance/exit characteristic (D_6):** $D_6 = 1-(S_7/S_1)$.
3. Once all the intermediate values are calculated, DSQI is calculated by the following equation.

$$DSQI = \Sigma w_i D_i$$

Where

i = 1 to 6

$\Sigma w_i = 1$ (w_i is the weighting of the importance of intermediate values).

NOTES

Component-level Design Metrics

In conventional software, the focus of component-level design metrics is on the internal characteristics of the software components. The software engineer can judge the quality of the component-level design by measuring module cohesion, coupling and complexity. Component-level design metrics are applied after procedural design is final. Various metrics developed for component-level design are listed below.

- **Cohesion metrics:** Cohesiveness of a module can be indicated by the definitions of the following five concepts and measures.
  - o **Data slice:** Defined as a backward walk through a module, which looks for values of data that affect the state of the module as the walk starts
  - o **Data tokens:** Defined as a set of variables defined for a module
  - o **Glue tokens:** Defined as a set of data tokens, which lies on one or more data slice
  - o **Superglue tokens:** Defined as tokens, which are present in every data slice in the module
  - o **Stickiness:** Defined as the stickiness of the glue token, which depends on the number of data slices that it binds.
- **Coupling metrics:** This metric indicates the degree to which a module is connected to other modules, global data and the outside environment. A metric for module coupling has been proposed, which includes *data and control flow coupling*, *global coupling*, and *environmental coupling*.
  - o Measures defined for data and control flow coupling are listed below.

    d_i = total number of input data parameters

    c_i = total number of input control parameters

    d_0 = total number of output data parameters

    c_0 = total number of output control parameters
  - o Measures defined for global coupling are listed below.

    g_d = number of global variables utilized as data

    g_c = number of global variables utilized as control
  - o Measures defined for environmental coupling are listed below.

    w = number of modules called

    r = number of modules calling the modules under consideration

NOTES

By using the above mentioned measures, module-coupling indicator (m_c) is calculated by using the following equation.

$$m_c = K/M$$

Where

K = proportionality constant

$$M = d_i + (a*c_i) + d_0 + (b*c_0) + g_d + (c*g_c) + w + r.$$

Note that K, a, b, and c are empirically derived. The values of m_c and overall module coupling are inversely proportional to each other. In other words, as the value of m_c increases, the overall module coupling decreases.

- **Complexity metrics:** Different types of software metrics can be calculated to ascertain the complexity of program control flow. One of the most widely used complexity metrics for ascertaining the complexity of the program is cyclomatic complexity, which has already been discussed in Chapter 6.

User Interface Design Metrics

Many metrics have been proposed for user interface design. However, *layout appropriateness* metric and *cohesion metric* for user interface design are the commonly used metrics. **Layout Appropriateness (LA)** metric is an important metric for user interface design. A typical **Graphical User Interface** (**GUI**) uses many layout entities such as icons, text, menus, windows, and so on. These layout entities help the users in completing their tasks easily. In order to complete a given task with the help of GUI, the user moves from one layout entity to another. Appropriateness of the interface can be shown by absolute and relative positions of each layout entities, frequency with which layout entity is used, and the cost of changeover from one layout entity to another.

Cohesion metric for user interface measures the connection among the on-screen contents. Cohesion for user interface becomes high when content presented on the screen is from a single major data object (defined in the analysis model). On the other hand, if content presented on the screen is from different data objects, then cohesion for user interface is low.

In addition to these metrics, the direct measure of user interface interaction focuses on activities like measurement of time required in completing specific activity, time required in recovering from an error condition, counts of specific operation, text density, and text size. Once all these measures are collected, they are organized to form meaningful user interface metrics, which can help in improving the quality of the user interface.

Metrics for Object-oriented Design

In order to develop metrics for object-oriented (OO) design, nine distinct and measurable characteristics of OO design are considered, which are listed below.

NOTES

- **Complexity:** Determined by assessing how classes are related to each other
- **Coupling:** Defined as the physical connection between OO design elements
- **Sufficiency:** Defined as the degree to which an abstraction possesses the features required of it
- **Cohesion:** Determined by analyzing the degree to which a set of properties that the class possesses is part of the problem domain or design domain
- **Primitiveness:** Indicates the degree to which the operation is atomic
- **Similarity:** Indicates similarity between two or more classes in terms of their structure, function, behavior, or purpose
- **Volatility:** Defined as the probability of occurrence of change in the OO design
- **Size:** Defined with the help of four different views, namely, *population*, *volume*, *length*, and *functionality*. **Population** is measured by calculating the total number of OO entities, which can be in the form of classes or operations. **Volume** measures are collected dynamically at any given point of time. **Length** is a measure of interconnected designs such as depth of inheritance tree. **Functionality** indicates the value rendered to the user by the OO application.

Metrics for Coding

Halstead proposed the first analytic laws for computer science by using a set of primitive measures, which can be derived once the design phase is complete and code is generated. These measures are listed below.

n_1 = number of distinct operators in a program

n_2 = number of distinct operands in a program

N_1 = total number of operators

N_2 = total number of operands.

By using these measures, Halstead developed an expression for overall *program length*, *program volume*, *program difficulty*, *development effort*, and so on.

Program length (N) can be calculated by using the following equation.

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2.$$

Program volume (V) can be calculated by using the following equation.

$$V = N \log_2 (n_1+n_2).$$

Note that program volume depends on the programming language used and represents the volume of information (in bits) required to specify a program. Volume ratio (L) can be calculated by using the following equation.

$$L = \frac{\text{Volume of the most compact form of a program}}{\text{Volume of the actual program}}$$

Where, value of L must be less than 1. Volume ratio can also be calculated by using the following equation.

NOTES

$$L = (2/n_1)^* (n_2/N_2).$$

Program difficulty level (D) and effort (E) can be calculated by using the following equations.

$$D = (n_1/2)^*(N_2/n_2).$$

$$E = D * V.$$

Metrics for Software Testing

Majority of the metrics used for testing focus on testing process rather than the technical characteristics of test. Generally, testers use metrics for analysis, design, and coding to guide them in design and execution of test cases.

Function point can be effectively used to estimate testing effort. Various characteristics like errors discovered, number of test cases needed, testing effort, and so on can be determined by estimating the number of function points in the current project and comparing them with any previous project.

Metrics used for architectural design can be used to indicate how integration testing can be carried out. In addition, cyclomatic complexity can be used effectively as a metric in the basis-path testing to determine the number of test cases needed.

Halstead measures can be used to derive metrics for testing effort. By using program volume (V) and program level (PL), Halstead effort (e) can be calculated by the following equations.

$$e = V/PL$$

Where

$$PL = 1/[(n_1/2) \times (N_2/n_2)] \quad ...(1)$$

For a particular module (z), the percentage of overall testing effort allocated can be calculated by the following equation.

Percentage of testing effort (z) = e(z)/Σe(i)

Where, e(z) is calculated for module z with the help of equation (1). Summation in the denominator is the sum of Halstead effort (e) in all the modules of the system.

Metrics for Object-oriented Testing

For developing metrics for object-oriented (OO) testing, different types of design metrics that have a direct impact on the testability of object-oriented system are considered. While developing metrics for OO testing, inheritance and encapsulation are also considered. A set of metrics proposed for OO testing is listed below.

NOTES

- **Lack of cohesion in methods (LCOM):** This indicates the number of states to be tested. LCOM indicates the number of methods that access one or more same attributes. The value of LCOM is 0, if no methods access the same attributes. As the value of LCOM increases, more states need to be tested.
- **Percent public and protected (PAP):** This shows the number of class attributes, which are public or protected. Probability of adverse effects among classes increases with increase in value of PAP as public and protected attributes lead to potentially higher coupling.
- **Public access to data members (PAD):** This shows the number of classes that can access attributes of another class. Adverse effects among classes increase as the value of PAD increases.
- **Number of root classes (NOR):** This specifies the number of different class hierarchies, which are described in the design model. Testing effort increases with increase in NOR.
- **Fan-in (FIN):** This indicates multiple inheritances. If value of FIN is greater than 1, it indicates that the class inherits its attributes and operations from many root classes. Note that this situation (where FIN > 1) should be avoided.

Metrics for Software Maintenance

For the maintenance activities, metrics have been designed explicitly. IEEE have proposed **Software Maturity Index** (**SMI**), which provides indications relating to the stability of software product. For calculating SMI, following parameters are considered.

- Number of modules in current release (M_T)
- Number of modules that have been changed in the current release (F_c)
- Number of modules that have been added in the current release (F_a)
- Number of modules that have been deleted from the current release (F_d).

Once all the parameters are known, SMI can be calculated by using the following equation.

$$SMI = [M_T - (F_a + F_c + F_d)]/M_T.$$

Note that a product begins to stabilize as SMI reaches 1.0. SMI can also be used as a metric for planning software maintenance activities by developing empirical models in order to know the effort required for maintenance.

11.8 PROJECT METRICS

Project metrics enable the project managers to assess current projects, track potential risks, identify problem areas, adjust workflow, and evaluate the project

team's ability to control the quality of work products. Note that project metrics are used for tactical purposes rather than strategic purposes used by the process metrics.

NOTES

Project metrics serve two purposes. One, they help to minimize the development schedule by making necessary adjustments in order to avoid delays and alleviate potential risks and problems. Two, these metrics are used to assess the product quality on a regular basis and modify the technical issues if required. As the quality of the project improves, the number of errors and defects are reduced, which in turn leads to a decrease in the overall cost of a software project.

Applying Project Metrics

Often, the first application of project metrics occurs during estimation. Here, metrics collected from previous projects act as a base from which effort and time estimates for the current project are calculated. As the project proceeds, original estimates of effort and time are compared with the new measures of effort and time. This comparison helps the project manager to monitor (supervise) and control the progress of the project.

As the process of development proceeds, project metrics are used to track the errors detected during each development phase. For example, as software evolves from design to coding, project metrics are collected to assess quality of the design and obtain indicators that in turn affect the approach chosen for coding and testing. Also, project metrics are used to measure production rate, which is measured in terms of models developed, function points, and delivered lines of code.

11.9 MEASURING SOFTWARE QUALITY

The aim of the software developer is to develop high-quality software within a specified time and budget. To achieve this, software should be developed according to the functional and performance requirements, document development standards, and characteristics expected from professionally developed software. Note that private metrics are collected by software engineers and then assimilated to achieve project-level measures. The main aim at the project level is to measure both the errors and defects. These measures are used to derive metrics, which provide an insight into the efficacy of both individual and group software quality assurance and software control activities.

Many measures have been proposed for assessing software quality such as interoperability, functionality, and so on. However, it has been observed that reliability, correctness, maintainability, integrity, and usability are most useful as they provide valuable indicators to the project team.

- **Reliability:** The system or software should be able to maintain its performance level under given conditions. Reliability can be defined as the

NOTES

ability of the software product to perform its required functions under stated conditions for a specified period of time or for a specified number of operations. Reliability can be measured using **Mean Time Between Failure** (**MTBF**), which is the average of time between successive failures. A similar measure to MTBF is **Mean Time To Repair** (**MTTR**) which is the average time taken to repair the machine after a failure occurs. MTBF can be combined with **Mean Time To Failure** (**MTTF**), which describes how long the software can be used to calculate MTBF, that is,

MTBF = MTTF + MTTR.

- **Correctness:** A system or software must function correctly. Correctness can be defined as the degree to which software performs its specified function. It can be measured in terms of defects per KDLOC. For quality assessment, defects are counted over a specified period of time.
- **Maintainability:** In software engineering, software maintenance is one of the most expensive and time-consuming activities. Maintainability can be defined as the ease with which a software product can be modified to correct errors, to meet new requirements, to make future maintenance easier, or adapt to the changed environment. Note that software maintainability is assessed by using indirect measures like **Mean Time to Change** (**MTTC**), which can be defined as the time taken to analyze change request, design modifications, implement changes, testing, and distribute changes to all users. Generally, it has been observed that programs having lower MTTC are easier to maintain.
- **Integrity:** In the age of cyber-terrorism and hacking, software integrity has become an important factor in the software development. Software integrity can be defined as the degree to which unauthorized access to the components of software (program, data, and documents) can be controlled.

  For measuring integrity of software, attributes such as threat and security are used. **Threat** can be defined as the probability of a particular attack at a given point of time. **Security** is the probability of repelling an attack, if it occurs. Using these two attributes, integrity can be calculated by using the following equation.

  Integrity = Σ[1-(threat*(1-security))].
- **Usability:** Software, which is easy to understand and easy to use is always preferred by the user. Usability can be defined as the capability of the software to be understood, learned, and used under specified conditions. Note that software, which accomplishes all the user requirements but is not easy to use, is often destined to fail.

In addition to the afore-mentioned measures, lack of conformance to software requirements should be avoided as these form the basis of measuring

software quality. Also, in order to achieve high quality both explicit and implicit requirements should be considered.

Defect Removal Efficiency (DRE)

NOTES

Defect removal efficiency (DRE) can be defined as the quality metrics, which is beneficial at both the project level and process level. Quality assurance and control activities that are applied throughout software development are responsible for detecting errors introduced at various phases of SDLC. The ability to detect errors (filtering abilities) is measured with the help of DRE, which can be calculated by using the following equation.

$$DRE = E/(E + D)$$

Where

E = number of errors found before software is delivered to the user

D = number of defects found after software is delivered to the user.

The value of DRE approaches 1, if there are no defects in the software. As the value of E increases for a given value of D, the overall value of DRE starts to approach 1. With an increase in the value of E, the value of D decreases as more errors are discovered before the software is delivered to the user. DRE improves the quality of software by establishing methods which detect maximum number of errors before the software is delivered to the user.

DRE can also be used at different phases of software development. It is used to assess the software team's ability to find errors at each phase before they are passed on to the next development phase. When DRE is defined in the context of SDLC phases, it can be calculated by the following equation.

$$DRE_i = E_i/(E_i + E_{i+1})$$

Where

E_i = number of errors found in phase i

E_{i+1} = number of errors that were ignored in phase i, but found in phase i + 1.

The objective of the software team is to achieve the value of DRE_i as 1. In other words, errors should be removed before they are passed on to the next phase.

11.10 OBJECT-ORIENTED METRICS

Lines of code and functional point metrics can be used for estimating object-oriented software projects. However, these metrics are not appropriate in the case of incremental software development as they do not provide adequate details for effort and schedule estimation. Thus, for object-oriented projects, different sets of metrics have been proposed. These are listed below.

NOTES

- **Number of scenario scripts:** Scenario scripts are a sequence of steps, which depict the interaction between the user and the application. A number of scenarios is directly related to application size and number of test cases that are developed to test the software, once it is developed. Note that scenario scripts are analogous to use-cases.
- **Number of key classes:** Key classes are independent components, which are defined in object-oriented analysis. As key classes form the core of the problem domain, they indicate the effort required to develop software and the amount of 'reuse' feature to be applied during the development process.
- **Number of support classes:** Classes, which are required to implement the system but are indirectly related to the problem domain, are known as **support classes**. For example, user interface classes and computation class are support classes. It is possible to develop a support class for each key class. Like key classes, support classes indicate the effort required to develop software and the amount of 'reuse' feature to be applied during the development process.
- **Average number of support classes per key class:** Key classes are defined early in the software project while support classes are defined throughout the project. The estimation process is simplified if the average number of support classes per key class is already known.
- **Number of subsystems:** A collection of classes that supports a function visible to the user is known as a **subsystem**. Identifying subsystems makes it easier to prepare a reasonable schedule in which work on subsystems is divided among project members.

The afore-mentioned metrics are collected along with other project metrics like effort used, errors and defects detected, and so on. After an organization completes a number of projects, a database is developed, which shows the relationship between object-oriented measure and project measure. This relationship provides metrics that help in project estimation.

11.11 ISSUES IN SOFTWARE METRICS

Implementing and executing software metrics is a cumbersome task as it is difficult to manage the technical and human aspects of the software measurement. Also, there exist many issues which prevent the successful implementation and execution of software metrics. These issues are listed below.

- **Lack of management commitment:** It is observed that management is not committed towards using software metrics due to the following reasons.
  - o Management opposes measurement.
  - o Software engineers do not measure and collect data as management does not realize their importance.

- Management charters a metrics program, but does not assist in deploying the program into practice.

NOTES

- **Collecting data that is not used:** Data collected during the measurement process should be such that it can be used to enhance the process, project, or product. This is because collecting incorrect data results in wrong decision making, which in turn leads to deviation from the software development plan.
- **Measuring too much and too soon:** In a software project, sometimes excess data is collected in advance, which is difficult to manage and analyze. This results in unsuccessful implementation of the metrics.
- **Measuring the wrong things:** Establishing metrics is a time consuming process and only those data that provide valuable feedback should be measured in an effective and efficient manner. To know whether data needs to be measured, a few questions should be addressed (if answers are no, then metrics should not be established).
  - Do data items collected relate to the key success strategies for business?
  - Are managers able to obtain the information they need to manage projects and people on time?
  - Is it possible to conclude from data obtained that process changes are working?
- **Imprecise metrics definitions:** Vague or ambiguous metrics definition can be misinterpreted. For example, some software engineers may interpret a software feature as unnecessary while some software engineers may not.
- **Measuring too little, too late:** Measuring too less, provides information, which is of little or no importance for the software engineers. Thus, software engineers tend to offer resistance in establishing metrics. Similarly, if data is collected too late, the requested data item may result in unnecessary delay in software project as project managers and software engineers do not get the data they need on time.
- **Misinterpreting metrics data:** Interpretation of metrics data is important to improve the quality of software. However, software metrics are often misinterpreted. For example, if the number of defects in the software increases despite effort taken to improve the quality, then software engineers might conclude that software improvement effort are doing more harm than good.
- **Lack of communication and training:** Inadequate training and lack of communication results in poor understanding of software metrics and measurement of unreliable data. In addition, communicating metrics data in an ineffective manner results in misinterpretation of data.

In order to resolve or avoid these issues, the purpose for which data will be used should be clearly specified before the measurement process begins. Also, project managers and software engineers should be adequately trained and

measured data should be properly communicated to them. Software metrics should be defined precisely so that they work effectively.

NOTES

Check Your Progress

1. What is software measurement?
2. Define software metric.
3. What is process metric?
4. What is the use of object oriented metrics?

11.12 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Software measurement is a quantified attribute of a characteristic of a software product or the software process.
2. A software metric is a standard of measure of a degree to which a software system or process possesses some property.
3. Process metrics are an invaluable tool for companies wanting to monitor, evaluate, and improve their operational performance across the enterprise.
4. The metrics for object oriented system focus on measurements that are applied to the class and the design characteristics. Object oriented metrics are usually used to assess the quality of software designs.

11.13 SUMMARY

- Software measures are collected throughout the software development life cycle with an intention to improve the software process on a continuous basis.
- Software metrics help project managers to gain an insight into the efficiency of the software process, project, and product.
- Software parameters are measured: process measurement through process metrics, product measurement through product metrics, and project measurement through project metrics.
- Software process metrics, software engineers are able to assess the efficiency of the software process that is performed using the process as a framework.
- Product metrics help software engineer to detect and correct potential problems before they result in catastrophic defects.
- Project metrics enable the project managers to assess current projects, track potential risks, identify problem areas, adjust workflow, and evaluate the project team's ability to control the quality of work products.

NOTES

- The aim of the software developer is to develop high-quality software within a specified time and budget.
- Lines of code and functional point metrics can be used for estimating object-oriented software projects.
- Implementing and executing software metrics is a cumbersome task as it is difficult to manage the technical and human aspects of the software measurement.

11.14 KEY WORDS

- **Process Metrics:** The metrics that enable software engineers to assess the efficacy of the software process that is performed using the process as a framework.
- **Project Metrics:** The metrics that illustrate the project characteristics and its execution.
- **Reliability:** The ability of the software product to perform its required functions under stated conditions for a specified period of time, or for a specified number of operations.

11.15 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. What is software metrics?
2. Differentiate between measures, metrics and indicators.
3. What are the different types of software metrics?

Long Answer Questions

1. What are the guidelines for software metrics?
2. Explain the steps for designing software metrics.
3. Explain the measures for software quality.

11.16 FURTHER READINGS

Pressman, Roger S. 1997. *Software Engineering, a Practitioner's Approach*. New Delhi: Tata McGraw-Hill.

Jalote, Pankaj. 1991. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

NOTES

Schach, Stephen R. 2005. *Object Oriented and Classical Software Engineering*. New Delhi: Tata McGraw-Hill.

Somerville, Ian. 2001. *Software Engineering*. New Delhi: Pearson Education.

Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandriolli . 1991. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hill of India.

Jawadekar, Waman S. 2004. *Software Engineering: Principles and Practice*. New Delhi: Tata McGraw-Hill.

BLOCK - V
RISK AND QUALITY MANAGEMENT

NOTES

UNIT 12 RISK STRATEGIES

Structure

12.0 INTRODUCTION

In this unit, you will learn about the risk strategies and risk identification. *Risk* is an expectation of loss, a potential problem that may or may not occur in the future. Risk strategy is a structured and coherent plan to identify, assess and manage risk.

12.1 OBJECTIVES

After going through this unit, you will be able to:

- Discuss reactive and proactive risk strategies
- Define software risk
- Identify risk

12.2 REACTIVE VS PROACTIVE RISK STRATEGIES

Risk is an unexpected event which produces an adverse effect on process of software development. Strategy is a plan of action formulated to reduce or avoid the effects of risks. In this section, you will learn about the reactive strategies and proactive strategies. Reactive risk strategies responds to unexpected events after their occurrence while proactive risk strategies helps in eliminating problems before the chances of occurrence.

NOTES

Difference between proactive and reactive strategies

Reactive Risk Strategies	Proactive Risk Strategies
Responds to the risk after occurrence.	Helps in eliminating the risk before it occurs.
Once hazard occurs, employees take action to prevent an accident.	Hazard mechanisms and threats are identified before hazard occurrence.
Examples are MOR, incident report and accident report.	Examples audits/inspections, voluntary reporting and surveys

12.3 SOFTWARE RISK AND RISK IDENTIFICATION

Software Risk is the possibility of suffering from loss in software development process.

Risk is an unexpected event that produces an adverse effect during software development process. It is the combination of constraints and uncertainties. Risks indicate possible future happenings and are not concerned with their effect that has been observed in the past. They are identified by the following attributes.

- **Probability that an event will occur:** Events can occur at any time during software development process or when the software is developed. For example, an event can occur when software developed on one computer system is transferred to another computer system. Here, both the computer systems can create incompatibility in the hardware or software. This incompatibility causes an event to occur and is identified as risk.
- **Loss associated with the event:** The adverse impact of an event can be loss of time, loss of money, and lack of expected quality. For example, there can be change in user requirements after the coding phase is complete. The change in user requirements results in loss of control when team members are developing the software according to earlier requirements of user.

Note that there is no fixed time for the occurrence of risks. Hence, to keep track of risks, risk management needs to be carried out throughout the software development process. Risk management is a systematic process, which focuses on identification, control, and elimination of risks, which impacts the resources of the organization and the project. Its objective is to determine the loss before risks occur and then determine ways to prevent or avoid the adverse impact of risks on the project. Risk management depends on the number and complexity of risks. Based on this, the impact of risks can be low, medium, high, or very high.

Uncertainty and Constraint

As stated earlier, risks are a combination of uncertainties and constraints. In Figure 12.1, the curved line indicates the acceptable level of risk depending on the project. To minimize the risks, one or both of the constraint and uncertainty can be

minimized. Generally, it is observed that it is difficult to minimize constraint, so uncertainty is reduced. Note that it is difficult to achieve software in which all the risks are eliminated. Hence, it is essential to minimize the effect of risks as they cannot be eliminated completely. For this purpose, effective risk management is required.

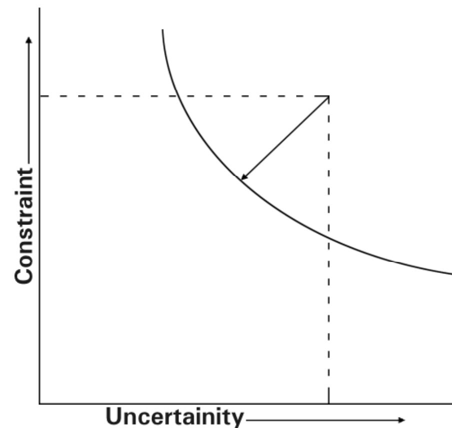


Fig. 12.1 Minimizing Risks in Projects

Principles of Risk Management

Every organization is subjected to risks. It is essential to manage the risks to prevent loss in an organization. If risks are not managed, it may result in project failure. Failure to manage risks is characterized by inability to do the following.

- Determine the measures used for managing risks
- Determine when risk management is required
- Identify whether risk management measures are taken.

Risk management is necessary because without elimination of risks, software cannot be developed properly. There are several principles that help in effective risk management. These principles are listed below.

- **Maintaining a global perspective:** Risks should be viewed in relation to software considering all the aspects of constraints and uncertainties in it. In addition, the impact of risks on business should be considered.
- **Having a forward-looking view:** Risks that may occur in future should be assumed.
- **Encouraging communication:** The experience of risks stated by project management team members or project manager should be considered.
- **Developing a shared software vision:** Both project management team and the senior management should be able to view the software and its risks with a common objective such as developing quality software and preventing loss. By following this approach, better risk identification and assessment is achieved.

- **Encouraging teamwork:** The skills and knowledge of every person involved in risk management should be combined when risk management activities are performed.

NOTES

Risk Management Plan

The risk management plan describes the process of assessing risks that may occur during software development. Risk management is concerned with determining the outcomes of risks before they happen and specifying procedures to avoid them and minimize their impact. It is important to consider the risks associated with a project as it helps in determining the ways to manage them when they occur. This in turn saves time and effort required for the software development.

Figure 12.2 shows risk management plan which comprises the following sections.

- **Statement:** Describes the purpose and advantage of identifying and recording risks
- **Objectives:** Describe the activities and procedures that are followed to manage risks
- **Roles and responsibilities:** Specify the roles and responsibilities of project management team, sponsor, and so on
- **Purpose**: Specifies the purpose of the process for risk management and the project library (database) where the risk management process is to be stored
- **Risk process:** Specifies the stages of risk management process and provides a process diagram to display risks for easier identification and management
- **Risk management worksheet:** Specifies the risk management worksheets such as risk management log in order to assess and control the risks

1.0 Statement
2.0 Objectives
3.0 Roles and Responsibilities
3.1 Process Task Name
3.2 User
3.3 Project Manager
3.4 Software Development Team
3.5 Sponsor
4.0 Purpose
5.0 Risk Process
6.0 Risk Management Worksheet
6.1 Risk management Log

Fig. 12.2 *Risk Management Plan*

Risk management comprises two activities, namely, *risk assessment* and *risk control*. These activities identify the risks and determine ways to eliminate them or reduce their impact.

Check Your Progress

1. What is reactive risk strategies?
2. What is software risk?
3. Define risk management.

NOTES

12.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Reactive risk strategies responds to unexpected events after their occurrence.
2. Software risk encompasses the probability of occurrence for uncertain events and their potential for loss within an organization.
3. Risk management is a systematic process, which focuses on identification, control, and elimination of risks, which impacts the resources of the organization and the project.

12.5 SUMMARY

- Risk is an unexpected event which produces an adverse effect on process of software development. Strategy is a plan of action formulated to reduce or avoid the effects of risks.
- Software risk encompasses the probability of occurrence for uncertain events and their potential for loss within an organization.
- Note that there is no fixed time for the occurrence of risks. Hence, to keep track of risks, risk management needs to be carried out throughout the software development process. Risk management is a systematic process, which focuses on identification, control, and elimination of risks, which impacts the resources of the organization and the project.
- Risks are a combination of uncertainties and constraints.
- The risk management plan describes the process of assessing risks that may occur during software development.
- Risk management comprises two activities, namely, risk assessment and risk control. These activities identify the risks and determine ways to eliminate them or reduce their impact

12.6 KEY WORDS

- **Risk:** It is an expectation of loss, a potential problem that may or may not occur in the future.

- **Software Risk:** It is the possibility of suffering from loss in software development process.

NOTES

12.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Write a note on reactive strategies.
2. Write a note on proactive strategies.
3. Discuss how will you minimize the risk.

Long Answer Questions

1. How will you identify the risk in software projects?
2. What are the principles of risk management?

12.8 FURTHER READINGS

Pressman, Roger S. 1997. *Software Engineering, a Practitioner's Approach*. New Delhi: Tata McGraw-Hill.

Jalote, Pankaj. 1991. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

Schach, Stephen R. 2005. *Object Oriented and Classical Software Engineering*. New Delhi: Tata McGraw-Hill.

Somerville, Ian. 2001. *Software Engineering*. New Delhi: Pearson Education.

Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandriolli . 1991. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hill of India.

Jawadekar, Waman S. 2004. *Software Engineering: Principles and Practice*. New Delhi: Tata McGraw-Hill.

NOTES

UNIT 13 RISK PROJECTION AND REFINEMENT

Structure

13.0 INTRODUCTION

In this unit, you will learn about the risk assessment and RMMM plan. Risk assessment is a method or process to identify the risk factors that have the potential to cause harm or loss. Risk management, mitigation and monitoring (RMMM) plan documents all work executed as a part of risk analysis and used by the project manager as a part of the overall project plan.

13.1 OBJECTIVES

After going through this unit, you will be able to:

- Explain the process of risk assessment
- Understand risk mitigation and monitoring
- Describe RMMM plan

13.2 RISK ASSESSMENT: RISK PROJECTION AND REFINEMENT

Risk assessment concentrates on determination of risks. The occurrence of risks depends on their nature, scope, and timing. **Nature of risks** specifies the problems that arise when risk occurs. **Scope of risks** specifies the capability of risks to impact the software. **Timing of risks** considers when and for how long the effect of risk is observed. With the help of proper risk assessment, risks can be prevented. It is important to provide information according to the level of risks. With complete information of risk assessment, the probability of occurrence of risks and its severity

NOTES

is determined. Generally, risk assessment can be carried out at any time during the project development. However, it is better to begin risk assessment as soon as possible in the development process. To effectively perform risk assessment, the management should consider the occurrence of risks instead of assuming that no risks will occur during software development. In addition, a record of risk assessment should be maintained. For understanding the severity of risks (that is, low, high, very high), earlier documents should be used as a reference and compared with the risks being assessed in the present software.

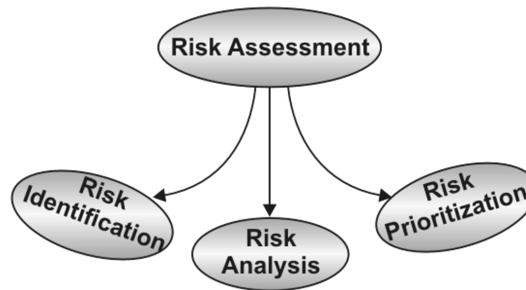


Fig. 13.1 *Risk Assessment*

Risk assessment comprises the following three functions (see Figure 13.1).

- **Risk identification:** Identifies the events that have an adverse impact on the project. These events can be a change in user requirements, new development technologies, and so on. In order to identify the risks, inputs from project management team members, users, and management are considered. The project plan including the sub-plans should also be carefully analyzed in order to identify the areas of uncertainty before starting the project. There are various kinds of risks that occur during software development. Some of the common types of risks are listed below.
  - o **Project risks:** These risks are harmful for project plan as they affect the project schedule and result in increased costs. They identify the budgetary, project schedule, personnel, stakeholder, and requirements problems.
  - o **Technical risks:** These risks are derived from software or hardware technologies, which are being used as a part in the software under development. These risks are harmful for the quality of a project and result in difficulty in implementation of software. They identify problems in design, implementation, interface, maintenance, and much more.
  - o **Business risks:** These risks are derived from the organizational environment where software is being developed. They are harmful for capability and existence of a project and result in difficulty by users to accept it. They identify market risk, sales risk, management risk, and so on.

- o **Known risks:** These risks are harmful as they are detected only after a careful examination of the project plan and result in technical problems. These risks include unrealistic delivery date of software, lack of project scope, and much more.
- o **Requirements risks:** These risks are derived from the change in user requirements. In addition, these risks occur when new user requirements are being managed.
- o **Estimation risks:** These risks are derived from the management estimations, which include the features required in the software and the resources required to complete the software.

- **Risk analysis:** Discovers possible risks using various techniques. These techniques include decision analysis, cost-risk analysis, schedule analysis, reliability analysis, and many more. After a risk is identified, it is evaluated in order to assess its impact on the project. Once the evaluation is done, risk is 'ranked' according to its probability of occurrence. After analyzing the areas of uncertainty, a description of how these areas affect the performance of project is made.
- **Risk prioritization:** Ranks the risks according to their capability to impact the project, which is known as **risk exposure**. Determination of risk exposure is done with the help of statistically-based decision mechanisms. These mechanisms specify how to manage the risk. The risks that impact the project the most should be specified so that they can be eliminated completely. On the other hand, the risks that are minor and have no or little impact on the project can be avoided.

13.3 RISK CONTROL, MITIGATION AND MONITORING

Risk control concentrates on the management of risks in order to minimize or eliminate their effect. For this purpose, risk control determines techniques and strategies to minimize the effect and resolve the factors that cause risks to occur. Risk control monitors the effectiveness of strategies used for risk control. In addition, it checks the change in severity of risks throughout the project. Note that risk control does not begin until risk assessment is complete. Once risk assessment is complete, risk control is performed to achieve an effective risk management. Risk control comprises the following three functions (see Figure 13.2).

- **Risk mitigation:** Minimizes the impact of risks. For this purpose, risk mitigation techniques are used, which are based on the occurrence of risks and their level of impact. Risk mitigation techniques incur additional cost on extra resources and time, which are required to complete the project. To avoid this, a cost-benefit analysis should be done to evaluate the benefits of eliminating the risks or minimizing their impact along with the costs required

NOTES

to implement risk management. Appropriate measures should be taken before the risks occur, which later resolves the effect of risks when the project is in progress.

- **Risk resolution:** Resolves the risks. Before resolving the risks, it is important to consider the areas of the project where risks can occur. After determining the areas, techniques are used for risk resolution. Generally, the techniques followed for risk resolution include cost/schedule estimates, quality monitoring, evaluation of new technologies, prototyping, and so on. The risks that have a very high impact should have an emergency plan. With the help of emergency plan, risks can be avoided or their impact can be minimized before they occur.
- **Risk monitoring:** Observes the risks of a project so that appropriate techniques can be applied when risks occur. The techniques used for risk monitoring include milestone tracking, tracking the risks that have greatest impact, continual risk re-assessment and so on. For risk monitoring, project management team members prioritize the risks and observe them. When risks are identified, their effects are recorded. With the help of these records, risks can be easily managed during the project.



Fig. 13.2 *Risk Control*

13.3.1 RMMM Plan

A risk management strategy can be defined as a software project plan or the risk management steps. It can be organized into a separate Risk Mitigation, Monitoring and Management Plan. The RMMM plan documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan. Teams do not develop a formal RMMM document. Rather, each risk is documented individually using a risk information sheet (RIS). In most cases, the RIS is maintained using a database system, so that creation and information entry, priority ordering, searches, and other analysis may be accomplished easily.

Check Your Progress

1. What is Risk control?
2. What does risk monitoring technique includes?
3. What does RMMM stands for?

13.4 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

1. Risk control concentrates on the management of risks in order to minimize or eliminate their effect. Risk control monitors the effectiveness of strategies used for risk control.
2. Risk monitoring technique includes milestone tracking, tracking the risks that have greatest impact, continual risk re-assessment. Project management team members prioritize the risks and observe them.
3. RMMM stands for Risk Mitigation, Monitoring and Management Plan.

13.5 SUMMARY

- Risk assessment concentrates on determination of risks. The occurrence of risks depends on their nature, scope, and timing. Nature of risks specifies the problems that arise when risk occurs.
- Risk control concentrates on the management of risks in order to minimize or eliminate their effect.
- Project risks are harmful for project plan as they affect the project schedule and result in increased costs.
- Risk mitigation minimizes the impact of risks. For this purpose, risk mitigation techniques are used, which are based on the occurrence of risks and their level of impact. Risk mitigation techniques incur additional cost on extra resources and time, which are required to complete the project.
- Risk monitoring include milestone tracking, tracking the risks that have greatest impact, continual risk re-assessment and so on.

13.6 KEY WORDS

- **Technical Risks:** These risks are derived from software or hardware technologies, which are being used as a part in the software under development.
- **Business Risks:** These risks are derived from the organizational environment where software is being developed.
- **Known Risks:** These risks are harmful as they are detected only after a careful examination of the project plan and result in technical problems.
- **Requirements Risks:** These risks are derived from the change in user requirements. In addition, these risks occur when new user requirements are being managed.

13.7 SELF ASSESSMENT QUESTIONS AND EXERCISES

NOTES

Short Answer Questions

1. What does the following terms specifies:
   (i) Nature of risk
   (ii) Scope of risk
   (iii) Timing of risk
2. Write a note on risk assessment.

Long Answer Questions

1. Explain the three functions of risk assessment.
2. Explain the three functions of risk control.

13.8 FURTHER READINGS

Pressman, Roger S. 1997. *Software Engineering, a Practitioner's Approach*. New Delhi: Tata McGraw-Hill.

Jalote, Pankaj. 1991. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

Schach, Stephen R. 2005. *Object Oriented and Classical Software Engineering*. New Delhi: Tata McGraw-Hill.

Somerville, Ian. 2001. *Software Engineering*. New Delhi: Pearson Education.

Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandriolli . 1991. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hill of India.

Jawadekar, Waman S. 2004. *Software Engineering: Principles and Practice*. New Delhi: Tata McGraw-Hill.

UNIT 14 QUALITY MANAGEMENT

NOTES

Structure

14.0 INTRODUCTION

Nowadays, quality has become an important factor to be considered while developing software. This is due to the fact that users are interested in quality software, which is according to their requirements and is delivered within a specified time. Furthermore, users require software, which is maintainable and minimizes the time and cost for correcting problems in it. The objective of the software development team is to design the software with minimum errors and required functionality according to user specified requirements. Software quality depends on various characteristics such as correctness, reliability, and efficiency. These characteristics are used as a checklist to implement quality in software.

14.1 OBJECTIVES

After going through this unit, you will be able to:

- Define software quality and software quality assurance
- Understand software reviews that are a systematic evaluation of the software in order to detect errors
- Explain formal technical review
- Discuss the evaluation of software quality
- Explain the process of software reliability

14.2 QUALITY CONCEPTS

NOTES

Quality refers to the features and characteristics of a product or service, which define its ability to satisfy user requirements. Generally, some concepts are used to develop high-quality software. These concepts give an overview of characteristics of the software in order to accomplish user requirements in it. Various quality concepts are listed below.

- Conformity with established standards and end user requirements
- Ease of use and operation
- Ease of modification of existing functions and addition of new functions
- Performance as expected by the user
- Well tested for functionality, user interface, and performance
- Well documented.

Software quality refers to the extent to which the software is able to perform correctly along with required features and functions. For this, a planned and systematic set of activities is performed. **IEEE** defines software quality as '*(1) the degree to which a system, component, or process meets specified requirements. (2) the degree to which a system, component, or process meets customer or user needs or expectations.*'

Principles followed for achieving good quality software are listed below.

- **Prevention of errors:** In order to avoid errors in the code, software tools are extensively used.
- **Detection and correction of errors:** Errors should be detected in the early stages of software development, so that much time is not consumed to correct them in the later stages. Thus, quality control should be carried out during all the phases of the SDLC.
- **Elimination of cause(s) of error:** After detection of errors, it is necessary to eliminate their cause. Eliminating causes of errors increases the quality of the software.
- **Independent audit:** The objective of an audit is to assess the product and process activities. Thus, independent audit for product and process should be conducted in accordance with the standards and procedures established in the quality process.

Software Quality Control

Software quality control is concerned with *product* quality and checks, whether the product meets user requirements and is developed in accordance with the established standards and procedures. In addition, it is used to accomplish the following objectives.

- Comparison of product quality procedures with the established standards
- Identification of errors in order to rectify them
- Focus on reviews and testing the product
- Identification of outputs generated by the product.

NOTES

Software Quality Assurance (SQA)

Software quality assurance is concerned with *process* quality and refers to planned and systematic sets of activities, which ensure that software life cycle processes and products conform to requirements, standards, and procedures. In addition, SQA assures that the standards, processes, and procedures are appropriate for the software project and are correctly implemented. Note that a high-quality product can be developed if there is an appropriate process quality. The objectives of SQA are listed below.

- To determine the software objectives to be accomplished
- To establish software plans, when the objectives are determined
- To monitor and adjust software plans to satisfy user requirements.

Statistical Software Quality Assurance

Statistical SQA is a technique which measures the quality using quantitative techniques. It means information about defects in collected and categorized and an attempt is made to trace defect to underlying cause.

Quality Costs

Costs incurred in performing quality-related activities are referred to as quality costs. These activities include preventing, finding, and correcting errors in the software. The quality costs are broadly divided into three categories (see Figure 14.1), which are listed below.

- **Prevention costs:** These costs are incurred while examining the activities that affect software quality during the software development. These costs are particularly incurred to prevent developing poor quality software. For example, preventive costs are used to avoid errors in coding and designing. In addition, it includes the cost incurred on correcting mistakes in user manuals, staff training, requirements analysis, and fault tolerant design of software.
- **Appraisal costs:** These costs are incurred in verifying or evaluating the product before the software is delivered to the user. For example, appraisal costs include cost on conducting code inspection, white box testing, black box testing, training testers, and beta testing.
- **Failure costs:** These costs are incurred when the product does not meet user requirements. For example, failure costs are incurred in fixing errors

NOTES

while dealing with user complaints. Failure costs are further divided into following categories (see Figure 14.1).

- **Internal failure costs:** These costs are incurred *before* the software is delivered to the user. When software is unable to perform properly due to errors in it, internal failure costs are incurred for missed milestones and the overtime done to complete these milestones within the specified schedule. For example, internal failure costs are incurred in fixing errors that are detected in regression testing.
- **External failure costs:** These costs are incurred *after* the software is delivered to the user. When errors are detected after software is delivered, a lot of time and money is consumed to fix the errors. For example, external failure costs are incurred on the modification of software, technical support calls, warranty costs, and other costs imposed by law.



Fig. 14.1 *Types of Quality Costs*

Note: The total quality cost incurred on the software is the sum of the costs shown in Figure 14.1.

ISO 9126 Quality Factors

Software is said to be of high-quality if it is free from errors and is according to user requirements. In order to obtain the desired quality, factors defined in ISO 9126 are followed. Figure 14.2 shows the factors responsible for software quality.

These factors, which act as a checklist to assess the quality of software (see Figure 14.3) are discussed here.

- **Functionality:** The capability of software to achieve the intended purpose using functions and their specified properties is referred to as functionality. The attributes of functionality are listed below.
  - **Suitability:** Ability of software to perform a set of functions that are used for specified tasks
  - **Accuracy:** Appropriateness of the results in accordance with the user requirements

- **Interoperability:** Ability of software to interact with other systems
- **Security:** Prevention of unauthorized access to programs in software.

- **Reliability:** The capability of software to maintain its performance level under stated conditions for a period of time is referred to as reliability. The attributes of reliability are listed below.
  - **Fault tolerance:** Ability of software to perform its functions when unexpected faults occur
  - **Maturity:** Ability of software to handle frequency of failures
  - **Recoverability:** Ability of software to recover data that is lost due to software failure.
- **Usability:** The capability of software to be easily operated and understood is referred to as usability. The attributes of usability are listed below.
  - **Understandability:** The ease with which software can be understood by the user
  - **Learnability:** The ease with which the functioning of software can be learnt by the user
  - **Operability:** The ease with which the software can be operated by the user.
- **Efficiency:** The capability of software to perform with optimal resources under specified conditions is referred to as efficiency. The attributes of efficiency are listed below.
  - **Time behaviour:** Response time required by software to perform a function
  - **Resource behaviour:** Resources required by software to perform its functions.
- **Maintainability:** The capability of software to make specific modifications, which include correcting, improving, or adapting the software is referred to as maintainability. The attributes of maintainability are listed below.
  - **Testability:** Ability of software to be easily tested and validated
  - **Changeability:** Ability of software to be modified for fault removal and environmental change
  - **Analyzability:** Ability of software to allow detection of cause of faults and failures
  - **Stability:** Ability of software to withstand the risks occurring due to modifications.
- **Portability:** The capability of software to be transferred from one hardware or software environment to another. The attributes of portability are listed below.

NOTES

NOTES

- o **Installability:** Effort with which software is installed in a specified environment (both hardware and software environment)
- o **Conformance:** Effort with which software complies with the standards that are related to portability
- o **Replaceability:** Effort with which software is used in place of other software for a particular environment
- o **Adaptability:** Ability of software to adapt to a new environment or new technology.

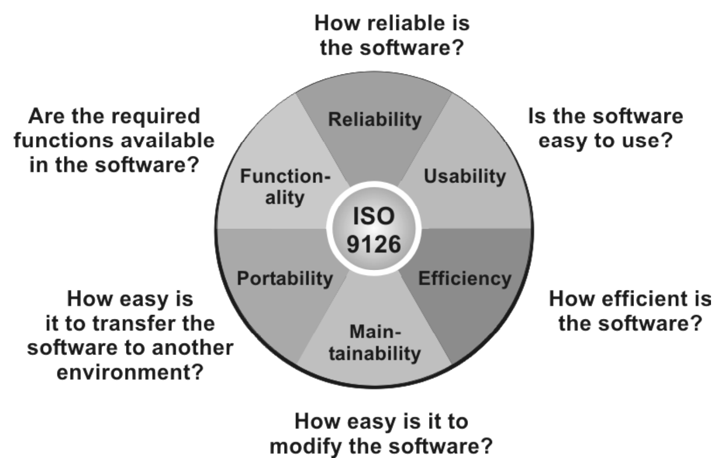


Fig. 14.2 *ISO 9126 Quality Factors*



Fig. 14.3 *ISO 9126 Quality Sub-attributes*

McCall's Quality Factors

There are several factors which are used as a measure to enhance quality of software. These factors are referred to as McCall's quality factors. As shown in Figure 14.4, these factors are characterized according to the following three important aspects.

- **Product operation:** Checks whether the software is able to perform the desired function efficiently. It comprises the following factors.

- o **Correctness:** The effort with which software programs perform according to user requirements
- o **Reliability:** The effort with which the software is expected to perform its specified functions accurately
- o **Usability:** The effort with which software programs are understood and operated
- o **Integrity:** The degree to which unauthorized access to the software can be controlled
- o **Efficiency:** The amount of software code and computing resources required by software programs to carry out specified functions.



Fig. 14.4 McCall's Quality Factors

- **Product revision:** Checks whether the software requires effort to modify it. It comprises the following factors.
  - o **Maintainability:** The effort with which software can be easily maintained so that less time is taken to fix errors
  - o **Flexibility:** The effort with which a software program is modified
  - o **Testability:** The effort required to test a software program to ensure that it performs the required functions.
- **Product transition:** Checks whether the software is adaptable to the new environment. It comprises the following factors.
  - o **Portability:** The effort required to transfer a software program from one hardware/software environment to another
  - o **Reusability:** The degree to which a software program can be reused in other applications depending on the scope and functions performed by other software programs
  - o **Interoperability:** The effort with which a system integrates with another system.

14.3 SOFTWARE QUALITY ASSURANCE GROUP

NOTES

Software quality assurance (SQA) comprises various tasks that are responsible for ensuring quality. These tasks are assigned to software engineers and the SQA group. The common task of software engineers and the SQA group is to verify that the software is developed according to the user requirements and established standards.

SQA group comprises the *quality head*, *quality analyst*, and *quality control staff*. This group is concerned with keeping track of responsibilities of each team member and ensuring that tasks are being performed properly so that high-quality software is developed. The **quality head** is responsible to check that all the individuals of SQA group are performing the assigned tasks correctly. The **quality analyst** is responsible for establishing plans and procedures for evaluating software quality. Furthermore, the quality analyst reviews the software project and ensures that software quality is according to the established standards. **Quality control staff** is responsible for checking the functionality and performance of software by executing it and conducting different kinds of testing to detect errors.

Various functions performed by the SQA group are discussed here.

- **Preparation of SQA plan:** This plan is made in the initial stages of development (during process planning) and is reviewed by the management who is responsible for making decisions and policies in an organization as well as assigning and controlling tasks to be performed. SQA plan determines the tasks that are to be carried out by software engineers and the SQA group. In addition, it identifies the audits and reviews to be performed and standards and documents to be produced by the SQA group.
- **Participation in the software development process description:** A process is chosen by the software developers while developing the software. The SQA group assists the software development team by reviewing the process description and checking whether it is in accordance with the user requirements and established standards. The choice of process depends on the complexity of the software project.
- **Reviewal of activities in compliance with the defined software process:** The SQA group examines the process in order to record the changes made to it. In addition, it ensures that project documentation is updated to accommodate accurate records, to further ensure proper understanding during software maintenance. With incorrect records, the process cannot be evaluated accurately.
- **Verification of compliance with defined standards:** The SQA group assesses the process and compares it with established procedures and standards so that quality is incorporated in the software during each phase of software development.

Software Quality Assurance Plan

NOTES

Quality planning is a structured process for defining the procedures and methods, which are used to develop software. Quality planning starts in the early phases of software development. The SQA plan defines the activities for the SQA group. The objective of the SQA plan is to identify the quality assurance activities that are followed by the SQA group. Generally, the plan comprises documentation such as project plan, test plans and user documents. Figure 14.5 shows an SQA plan designed by IEEE.

1.0 Introduction
    1.1 Scope and intent of SQA activities
    1.2 SQA organisational role
2.0 SQA tasks
    2.1 Task overview
        2.1.1 Description of SQA task management
        2.1.2 Work products and documentation
    2.2 Standards, practices and conversations (SPC)
    2.3 SQA resources
3.0 Reviews and audits
    3.1 Generic review guidelines
        3.1.1 Conducting a review
        3.1.2 Roles and responsibilities
        3.1.3 Review work products
    3.2 Formal technical reviews
        3.2.1 System specification review
        3.2.2 Software project plan review
        3.2.3 RMMM review
        3.2.4 Requirements reviews (models, specification)
        3.2.5 Data design review
        3.2.6 Architectural design review
        3.2.7 Interface (GUI) design review
        3.2.8 Component design review(s)
        3.2.9 Code reviews
        3.2.10 Test specification review
        3.2.11 Change control reviews and audits
    3.3 SQA audits
4.0 Problem reporting and corrective action/follow-up
    4.1 Reporting mechanisms
    4.2 Responsibilities
    4.3 Data collection and evaluation
    4.4 Statistical SQA
5.0 Software process improvement activities
    5.1 Goal and objectives of SPI
    5.2 SPI tasks and responsibilities
6.0 Software configuration management overview
7.0 SQA tools, techniques, methods
8.0 Appendix

Fig. 14.5 *SQA Plan*

This plan comprises eight sections. The description of each section is discussed here.

- **Introduction:** Provides an overview of the entire SQA plan and SQA activities being performed in the software project. In addition, it specifies the role of organization in SQA.
- **Tasks:** Provides information about the SQA tasks performed in the SQA plan. It assigns SQA tasks to the software engineers and the SQA group. In addition, it describes SQA resources such as hardware, software, and tools to perform SQA tasks.
- **Reviews and audits:** Describes project reviews conducted by the SQA group. For performing reviews and audit, a set of guidelines is used. Furthermore, it provides information on formal technical reviews. Audits are used to assess SQA activities conducted in the project.
- **Problem reporting and corrective action/follow-up:** Describes problem-reporting mechanisms that occur as a result of reviews and provides corrective actions to improve them. Reporting mechanisms include the process of reporting problems (such as errors in software) to SQA group. In addition, it collects erroneous data and finds measures to correct errors.

NOTES

- **Software process improvement activities:** Describes the activities required for Software Process Improvement (SPI). It also outlines the objectives and tasks of SPI.
- **Software configuration management (SCM) overview:** Provides an overview of SCM plan. This plan provides information such as description of configuration management process and procedures to perform SCM.
- **SQA tools, techniques**, **and methods:** Describes the SQA tools, techniques, and methods that are used by the SQA group.
- **Appendix:** Provides additional information about SQA plan.

14.4 SOFTWARE QUALITY ASSURANCE ACTIVITIES

To ensure that the software under development is in accordance with established standards, SQA activities are followed. These activities are concerned with verification and validation of the software process being followed to accomplish the user requirements in the software. When SQA activities are complete, a report describing the results is presented to the management. If the report indicates that the software quality is not according to the user requirements then the management takes necessary actions to improve the quality of the software. Broadly, software quality assurance activities are classified into *process quality assurance activity* and *product quality assurance activity*.

Process Quality Assurance Activity

This activity ensures that the processes that are defined for the project are followed. In case the defined processes are not followed according to standards, suggestions for those situations are provided to the management. A separate report is sent to both the management and the SQA group. The SQA group examines associated problems occurring due to delay in schedule or increase in budget and reports them to the management. The process quality is evaluated through software process audits. These audits focus on checking the implementation of the defined processes in software. Software process audits are conducted by quality analysts from time to time. Various tasks involved in process quality assurance activity are listed below.

- Ensuring development of project plans and their review and approval
- Conducting software configuration reviews
- Establishing software quality plans, standards, and procedures for the project
- Establishing periodic reviews with the user

- Submitting a profile to the management about the adherence of defined process throughout the software development process, by showing regular improvement in the development process
- Ensuring regular submission of metrics and other required information to the software engineering process group (SEPG).

NOTES

Product Quality Assurance Activity

This activity identifies the processes in order to check whether they are carried out properly to develop a product that conforms to the user requirements. Product quality assurance activity concentrates on detection and elimination of errors as early as possible during software development. As a result, there is reduction in costs incurred during software maintenance and testing. Product quality assurance comprises methods to conduct reviews and testing of the software. Reviews evaluate the software before delivering it to the user. After software reviews are complete, the quality analyst performs a final review. Final review is essential to verify that all planned reviews or reported problems are followed. In addition to reviews, software testing is performed to check errors in software and to verify whether the errors are occurring due to non-conformance to the user requirements.

Auditing in Software Quality Assurance Activities

SQA activities follow several techniques to find errors in the software. Audit is a fundamental technique used for both process quality assurance and product quality assurance. It examines a process or product in detail and compares it with the established standards. Audit is performed to examine the conformance of a development process to procedures and standards. In addition, it examines the records according to the defined process to determine whether the procedures are followed correctly. The auditor performs audit to examine whether the records, procedures, and management policies are executed properly. An auditor can be internal—one of the employees of the organization—as well as external, which is selected by the organization and SEPG.

14.5 SOFTWARE REVIEWS

Software reviews are systematic evaluation of the software in order to detect errors. These reviews are conducted in various phases of the software development process such as analysis, design, and coding. Note that software reviews are conducted for the product, software, or documentation. The objective of software reviews is to ensure that the quality of software is maintained in each phase of the software development. The process commonly followed for conducting software reviews is shown in Figure 14.6.

NOTES

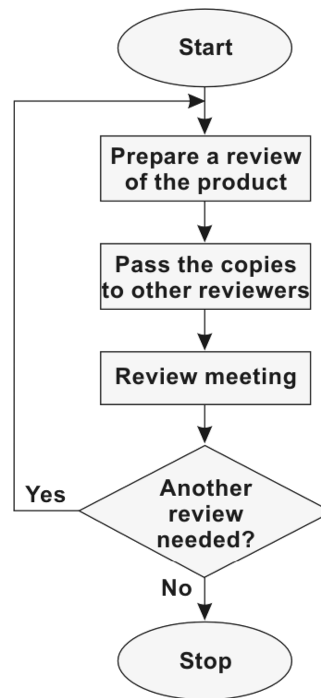


Fig. 14.6 Review Process

A review team performs software reviews (see Figure 14.7). The team members that constitute a review team are listed below.

- **Author or producer:** Develops the product and is responsible for making corrections in it after the review is over. The author raises issues concerning the product during review meeting.
- **Moderator or review leader:** The review leader performs the following activities.
  - o Ensures that review procedures are performed throughout the review meeting
  - o Ensures that the review team is performing their specified responsibilities
  - o Verifies the product for review
  - o Assembles an effective team and keeps review meetings on schedule
  - o Organizes a schedule of the product to be reviewed.
- **Recorder:** Records important issues of the product, which arise during a review meeting. A document is prepared containing the results of the review meeting. This document includes type and source of errors.
- **Reviewer or inspector:** Analyzes the product and prepares notes before the review meeting begins. There can be more than one reviewer in the team depending upon the size of the product to be reviewed.

NOTES



Fig. 14.7 *Review Team*

Different types of reviews are conducted as a part of the review process. The software review can be conducted in both formal and informal ways. A review meeting can be conducted in a cafeteria with the purpose of solving technical problems. Similarly, a meeting held where some agenda is discussed formally is also a form of software review. Generally, software reviews are of two types, namely, *walkthroughs* and *formal technical reviews*.

Walkthroughs

This review is an informal review, which is used to evaluate the product. No preparation is made in advance and due to this, walkthroughs are termed as an unplanned activity. In a walkthrough, the author describes the product to the peers (group of individuals other than the author) and gets feedback in an informal way (see Table 14.1). Note that the people involved in this review do not have any specified responsibility.

Table 14.1 *Tasks in a Walkthrough*

Tasks	***Evaluators***
Pass the product to the peer for a walkthrough.	Author
Suggestions for improving the software are presented to the author.	Peer members
Implement necessary rework on the product according to the suggestions of peer members.	Author

As walkthroughs are informal, they are not considered efficient. This is because walkthroughs leave several errors unnoticed. In addition, no record of errors is maintained. It results in difficulties in verifying the correctness of the product after walkthroughs are 'over'.

Formal Technical Review (FTR)

NOTES

A formal technical review (FTR) is a formal review that is performed by a review team. It is performed in any phase of software development for any work product, which may include requirements specification, design document, code, and test plan. Each FTR is conducted as a meeting and is considered successful only if it is properly planned, controlled, and attended. The objectives of FTR are listed below.

- To detect errors in functioning of software and errors occurring due to incorrect logic in software code
- To check whether the product being reviewed accomplishes user requirements
- To ensure that a product is developed using established standards.

Review Guidelines

To conduct a formal technical review, there are some review guidelines. These guidelines are established before a review meeting begins. The commonly followed review guidelines are listed below.

- **Review the product:** The focus of formal review should be to detect errors in the product instead of pointing out mistakes (if any) committed by a team member. The aim should be to conduct the review in harmony among all the team members.
- **Set the agenda:** Formal reviews should keep a track of the schedule. The moderator is responsible for maintaining this schedule. For this purpose, he ensures that each review team member is performing the assigned task properly.
- **Keep track of discussion:** During the review meeting, various issues arise and it is possible that each review team member has a different view on an issue. Such issues should be recorded for further discussion.
- **Advance preparation:** Reviewers should make an advance preparation for the product to be reviewed. For this purpose, the reviewers should note the issues that can arise during the review meeting. Then, it is easy for them to discuss the issues during the review meeting.
- **Indicate problems in the product:** The objective of a review meeting should be only to indicate the problems or errors. In case there are no proper suggestions for the problems, a review meeting should be conducted again.
- **Categorize the error:** The errors detected in the software should be classified according to the following categories.
    - o **Critical errors:** Refer to errors that bring the execution of the entire software to a halt. Thus, crucial errors need to be 'fixed' before the software is delivered.

NOTES

- o **Major errors:** Refer to errors that affect the functionality of programs during their execution. Like critical errors, major errors need to be fixed before the software is delivered
- o **Besides errors:** Refer to errors that do not affect the usability of the software
- o **No errors:** Indicates that there are no errors in the software.

- **Prepare notes:** The recorder who is one of the reviewers should keep a record of the issues in order to set priorities for other reviewers as the information is recorded.
- **Specify the number of people:** There should be a limited number of individuals in the meeting that should be specified before the meeting begins.
- **Develop a checklist:** A checklist should be maintained at the end of the meeting. The checklist helps the reviewer to focus on important issues that are discussed at the meeting. Generally, a checklist should be prepared for analysis, design, and code documents.

Review Meeting

A review meeting is conducted to review the product in order to validate its quality. Review team members examine the product to identify errors in it. As shown in Figure 14.8, a successful review consists of a number of stages, which are described here.

- **Planning:** In this stage, the author studies the product that requires a review and informs the moderator about it. The moderator verifies the product that is to be examined. The verification is essential to determine whether the product requires a review. After verification, the moderator assigns tasks to the review team members. The objective of planning is to ensure that the review process is followed in an efficient manner. In addition, it ensures that a proper schedule is made to conduct an effective review.
- **Overview:** In this stage, the product is analyzed in order to detect errors. For this purpose, knowledge of the product is essential. In case reviewers do not have proper knowledge of the product, the author explains the functionality and the techniques used in the product.
- **Preparation:** In this stage, each review team member examines the product individually to detect and record problems (such as errors and defects) in the product. There are several types of problems that are identified during the preparation stage. Generally, the problems considered at this stage are listed below.
  - o **Clarity:** User requirements are not understood properly
  - o **Completeness:** Details of user requirements are incomplete
  - o **Consistency:** Names of data structures and functions are used illogically

NOTES

- o **Functionality:** Functions, inputs, and outputs are not specified properly
- o **Feasibility:** Constraints such as time, resources, and techniques are not specified correctly.

- **Meeting:** In this stage, the moderator reviews the agenda and issues related to the product. The problems described in the overview stage are discussed among review team members. Then, the recorder records the problems in a defect list, which is used to detect and correct the errors later by the author. The defect list is divided into the following categories.
  - o **Accept the product:** In this category, the author accepts the product without the need for any further verification. This is because there are no such problems that halt the execution of the product.
  - o **Conditionally accept the product:** In this category, the author accepts the product, which requires verification. If there are problems in the product, the next stage (rework stage) is followed.
  - o **Re-examine the product:** In this category, the author re-examines the product to understand the problems in it. After rework, the product is sent to the moderator again to verify that problems are eliminated.
- **Rework:** In this stage, the author revises the problems that are identified during the review meeting. He determines the problems in the product and their causes with the help of a defect list. Then, he resolves the problems in the product and brings it back to the moderator for the follow-up stage.
- **Follow-up:** In this stage, the product is verified after the author has performed rework on it. This is due to the fact that he may have introduced new errors in the product during rework. In case there are still some errors in the product, a review meeting is conducted again.

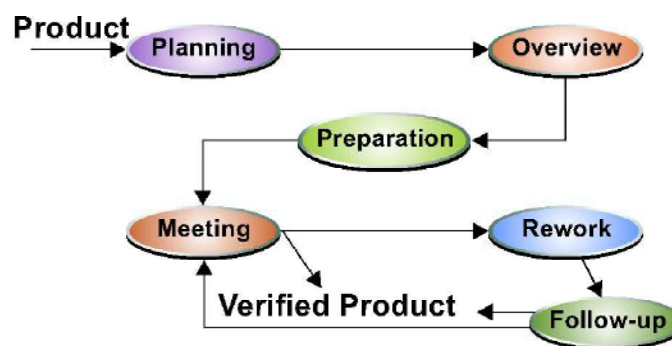


Fig. 14.8 *Stages of Review Meeting*

While the review is being conducted, the recorder records all the issues raised during the review meeting. After he completes the task, the reviewer summarizes the review issues. These issues are recorded in two kinds of documents, which are listed below.

NOTES

- **Review Issue list:** This document is concerned with the identification of problems in the product. It also acts as a checklist that informs the author about the corrections made in the product.
- **Review summary report:** This document focuses on information such as the phase of software development that is reviewed, the review team member who reviewed it, and conclusions of the review meeting. Generally, a review summary report comprises a single page and is advantageous for future reference by the software development team.

Cost Impact of Software Errors

Formal technical reviews are used for detecting errors during the development process of the software. This implies that reviews detect errors and thereby reduce the cost of software and increase its efficiency. Thus, the formal technical reviews provide cost-effective software.

For example, Figure 14.9 (a) shows the errors present in software before conducting a formal technical review. When FTR is conducted, there is reduction in errors, as shown in Figure 14.9 (b). Here, approximately half of the errors are fixed and some part of software still contains errors in it. These errors can occur when some of the errors are fixed and due to it, new errors arise in the software. It is observed that FTR conducted in early stages of the software development process reduces the time and cost incurred in detecting errors in software.

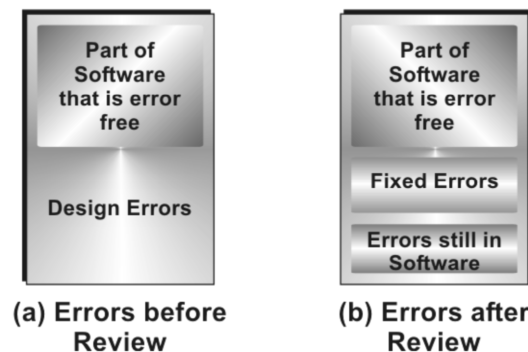


Fig. 14.9 *Detection of Errors*

14.6 EVALUATION OF QUALITY

Various kinds of approaches are used to improve quality of the software and overall process of the organization. The commonly used software quality approaches are *Six Sigma* and *ISO standards*.

Six Sigma

Today, Six Sigma is regarded as the most widely used strategy for evaluating quality. Six Sigma refers to six standard deviations. The standard deviation describes

NOTES

the distribution of the process or procedure used in software development. It is a statistical strategy to ensure quality assurance of processes in the organization. Thus, it can be stated that Six Sigma focuses on identification and prevention of errors in the software development process. In addition, it is used to eliminate errors in the software by evaluating the performance of the software process. The six standard deviations used in Six Sigma strategy are listed below.

- Good business strategy that balances cost, quality, features, and constraints as well as matches business priorities
- Decisions made are tied to bottom line of the organization
- Exercise care to use correct measurements in each situation
- Consider measuring output for a longer period
- Limited scope with adequate importance and reasonable time
- Clear quantitative measures to define success.

The objective of Six Sigma is to reduce variance (an activity that varies from a norm or a standard) and improve the processes in an organization. In addition, it minimizes the cost of poor quality. These objectives are directly associated with the objectives of an organization. Six Sigma monitors day-to-day activities of an organization, in order to make optimal use of resources. According to Six Sigma, processes in an organization can be improved by applying the following approaches.

- **DMAIC:** It stands for define, measure, analyze, improve, and control. This approach checks whether a process is performing correctly. It improves the system by improving the process that is not able to meet user requirements. Various attributes and functions present in this approach are listed in Table 14.2.

Table 14.2 *DMAIC Methodology*

Attributes	*Functions*
Define	Identify goals of the project and the user requirements.
Measure	Quantify the process to determine the current performance of the software.
Analyze	Examine and determine the cause of the error.
Improve	Improve the process by eliminating the errors.
Control	Determine future process performance for prevention of errors.

- **DMDAV:** It stands for define, measure, design, analyze, and verify. This approach is used while a software process is being developed in the organization. Various attributes and functions present in this approach are listed in Table 14.3.

NOTES

Table 14.3 DMDAV Methodology

Attributes	Functions
Define	Specify goals of the project and the user requirements.
Measure	Evaluate and determine the user requirements.
Design	Design the process to avoid the root cause of defect.
Analyze	Study process options to meet the user requirements.
Verify	Ascertain the design performance so that the user requirements are accomplished.

Note: In order to achieve Six Sigma, the errors should not be more than 3.4 per million occurrences.

ISO 9000 Quality Standards

The ISO 9000 standards describe the elements in quality assurance, which outline the requirements of good quality product in an organization. The elements comprise organizational structure, procedures, and resources that are needed to implement quality assurance in the software. These standards also provide auditing tools to make sure that they are properly implemented according to the standards and meet the user requirements. After implementing these standards, it is mandatory to audit the organization to evaluate its effectiveness. After a successful audit, the organization receives a registration certificate, which identifies the quality of the organization as being in compliance with ISO 9000 standards. ISO 9000 series includes three models, which are listed below.

- **ISO 9001:** This quality assurance model applies to organizations that design, develop, install, and service products. It discusses how to meet customer needs effectively.
- **ISO 9002:** This quality assurance model applies to organizations that produce, install, and service products. This model is nearly identical to 9001, except that it does not incorporate design and development.
- **ISO 9003:** This quality assurance model applies to organizations whose processes are almost exclusive to inspection and testing of final products. This model is currently limited to an inspection function.

One of the quality standards that is commonly used in practice is ISO 9001:2000. It is a quality management standard that is established for improving the overall quality of the process and product in an organization. It can be achieved by using appropriate management policies and practices. The ISO 9001:2000 is based on several management principles relating to different requirements. These principles are discussed here.

- **Management responsibility:** Describes a policy and regular reviews of management tasks and activities
- **Sales:** Provides information, which is helpful in understanding customer needs

NOTES

- **Document control:** Provides information on manuals and documentation prepared for an organization
- **Purchasing:** Provides information to customers on how to make sure that they are purchasing the required products
- **Identification and traceability:** Provides information on how to identify products and services (particularly certified items)
- **Inspection and test:** Provides information on how to test and inspect the products.
- **Corrective and preventive action:** Provides information on how to detect faults in the products and ways to manage them
- **Quality system:** Consists of a quality manual, which shows whether the manual applies to the management. It also examines the procedures and processes of the product.

14.7 SOFTWARE RELIABILITY

Software reliability is one of the quality factors that focus on the following tasks.

- Description of requirements for software, fault/ failure detection, and recovery of software in case of failure
- Review of software development process and products for software error prevention
- Description of processes for measuring and analyzing errors and description of reliability and maintainability factors for the software.

IEEE defines software reliability as '*the process of optimizing the reliability of software through a program that emphasizes software error prevention, fault detection and removal and the use of measurements to maximize reliability in light of project constraints, such as resources, schedule, and performance.*' Reliability constitutes two major activities, namely, *error prevention* and *fault detection and removal*.

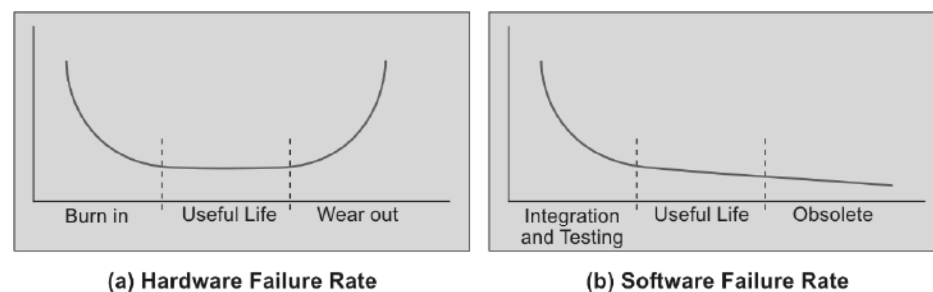


Fig. 14.10 *Hardware and Software Failure Rate*

A failure can be hardware failure or software failure. As shown in Figure 14.10 (a), hardware has three phases, namely, *burn in*, *useful life*, and *wear*

NOTES

out. In the burn in phase, the failure rate is high and decreases with time for the next phase, which is useful life. As compared to the burn in phase, the useful life phase is constant. The failure rate increases in the wear out phase. This is because the hardware components wear out with time. As shown in Figure 14.10(b), the rate of software failure as compared to Figure 14.10(a) is less and software failure occurs largely in the integration and testing phase. As the software is tested, errors are detected and removed. This results in decreasing the failure rate for the next phase of the software known as useful life. The software does not wear out, but becomes obsolete due to factors such as changes in the technology or the version, increase in complexity, and so on.

Three approaches are used to improve the reliability of software. These approaches are listed below.

- **Fault avoidance:** The design and implementation phase of the software development uses the process that minimizes the probability of faults before the software is delivered to the user.
- **Fault detection and removal:** Verification and validation techniques are used to detect and remove faults. In addition, testing and debugging can also remove faults.
- **Fault tolerance:** The designed software manages the faults in such a way that software failure does not occur. There are three aspects of fault tolerance.
  - **Damage assessment:** This detects the parts of software affected due to occurrence of faults.
  - **Fault recovery:** This restores the software to the last known safe state. Safe state can be defined as the state where the software functions as desired.
  - **Fault repair:** This involves modifying the software in such a way that faults do not recur.

Software Reliability Models

A software reliability model is examined as a mathematical analysis model for the purpose of measuring and assessing software quality and reliability quantitatively. As shown in Figure 14.11, detection and removal of errors incur huge costs. When the probability of software failure occurrence decreases, software reliability increases. A model used to describe software reliability is known as **Software Reliability Growth Model** (**SRGM**). The software reliability model evaluates the level of software quality before the software is delivered to the user. The objectives of software reliability models are listed below.

- To evaluate the software quantitatively
- To provide development status, test status, and schedule status
- To monitor reliability performance and changes in reliability performance of the software

NOTES

- To evaluate the maintenance cost for faults that are not detected during the testing phase.

The software reliability models can be divided into two classes. The first class deals with the designing and coding phases of software development by analyzing the reliability factors of the product. The second class deals with the testing phase. It describes the software failure-occurrence phenomenon or software fault-detection phenomenon by applying statistics theories that estimate software reliability. The software reliability models are broadly classified into two categories, namely, *dynamic models* and *static models*.

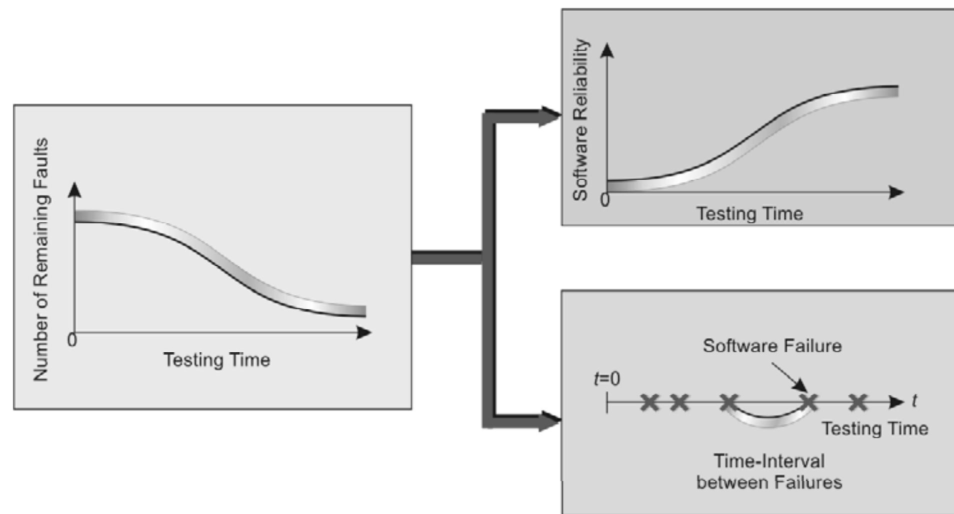


Fig. 14.11 *Software Reliability*

- **Dynamic models:** These models follow the changes of software during the testing phase. In addition, these models are based on the assumption that errors can be corrected as soon as they are detected. In other words, it is assumed that the time consumed in fixing errors is negligible.
- **Static models:** These models measure the quality of the software before debugging and testing.

Assessing Reliability and Availability

The measure of reliability is defined as the probability of software to operate without failures under given conditions. The ability to measure or predict reliability in software engineering focuses on three general purposes.

- Understanding and making trade-offs between reliability and other software characteristics such as performance, cost, and schedule
- Tracking progress when testing is complete
- Determining the effect of using new methods to develop software.

A common measure of how long the system can operate between failure of critical part is 'mean time between failure' (MTBF). In other words, MTBF is

defined as the average of times between successive failures. It is a statistical measure rather than a direct measure. A measure similar to MTBF is 'mean time to repair' (MTTR), which is the average amount of time taken to repair the machine after a failure occurs. It can be combined with 'mean time to failure' (MTTF), which describes how long the software can be used to calculate MTBF.

$$\text{MTBF} = \text{MTTF} + \text{MTTR}.$$

In addition to assessment of reliability, software availability is also calculated. Availability of software is defined as the probability of software to operate and deliver the desired request. Availability is an indirect measure to determine maintainability. This is because it determines the probability that software is working in accordance with the user requirements at a specified time. It is calculated from probabilistic measures MTBF and MTTR. In the mathematical notation, availability is defined as follows:

$$\text{Availability} = (1 - \text{MTTR} / (\text{MTTR} + \text{MTBF}))^{*}\ 100\%.$$

Software Safety

Software safety is a software quality assurance activity that follows a systematic approach to identify, analyze, record, and control software hazards to ensure that software operates in an intended manner. A hazard is defined as a set of conditions that can lead to an accident in a given set of conditions. These conditions are beyond the control of the software developer, that is, the software developer is unable to predict the conditions that can cause software to stop its functioning.

A software hazard occurs when a wrong input is given. Some hazards are avoidable and can be eliminated by changing the design of the system while others cannot be avoided and must be handled by the software. A technique known as fault-free analysis is used to prevent hazards in the developed software. In this analysis a detailed study is carried out to detect conditions that cause hazards. Once the hazards are analyzed, the requirements for the software are specified.

It is important to note the difference between software safety and software reliability. Software reliability uses statistical and mathematical methods to identify software failure and it is not necessary that this occurrence of failure leads to a hazardous situation. Software safety identifies the conditions in which these failures lead to a hazardous situation.

Check Your Progress

1. Define software quality assurance.
2. What is quality costs?
3. What are software reviews?
4. What is software reliability model?
5. Define formal technical review.

14.8 ANSWERS TO CHECK YOUR PROGRESS QUESTIONS

NOTES

1. Software quality assurance is concerned with process quality and refers to planned and systematic sets of activities, which ensure that software life cycle processes and products conform to requirements, standards, and procedures.
2. Costs incurred in performing quality-related activities are referred to as quality costs.
3. Software reviews are systematic evaluation of the software in order to detect errors.
4. A software reliability model is examined as a mathematical analysis model for the purpose of measuring and assessing software quality and reliability quantitatively.
5. A formal technical review (FTR) is a formal review that is performed by a review team.

14.9 SUMMARY

- Quality refers to the features and characteristics of a product or service, which define its ability to satisfy user requirements.
- Software quality control is concerned with product quality and checks, whether the product meets user requirements and is developed in accordance with the established standards and procedures.
- Software quality assurance is concerned with process quality and refers to planned and systematic sets of activities, which ensure that software life cycle processes and products conform to requirements, standards, and procedures.
- Costs incurred in performing quality-related activities are referred to as quality costs.
- Software quality assurance (SQA) comprises various tasks that are responsible for ensuring quality. These tasks are assigned to software engineers and the SQA group.
- Quality planning is a structured process for defining the procedures and methods, which are used to develop software. Quality planning starts in the early phases of software development.
- Software reviews are systematic evaluation of the software in order to detect errors. These reviews are conducted in various phases of the software development process such as analysis, design, and coding.

NOTES

- A formal technical review (FTR) is a formal review that is performed by a review team. It is performed in any phase of software development for any work product, which may include requirements specification, design document, code, and test plan.
- A review meeting is conducted to review the product in order to validate its quality. Review team members examine the product to identify errors in it
- Various kinds of approaches are used to improve quality of the software and overall process of the organization. The commonly used software quality approaches are Six Sigma and ISO standards.
- A software reliability model is examined as a mathematical analysis model for the purpose of measuring and assessing software quality and reliability quantitatively.
- Software safety is a software quality assurance activity that follows a systematic approach to identify, analyze, record, and control software hazards to ensure that software operates in an intended manner.

14.10 KEY WORDS

- **Software Quality Assurance (SQA):** A planned and systematic sets of activities, which ensure that software life cycle processes and products conform to requirements, standards, and procedures.
- **SQA Group:** A group comprising the quality head, quality analyst, and quality control staff, which is concerned with keeping track of responsibilities of each team member and ensuring that tasks are being performed properly so that high-quality software is developed.
- **Software Review:** The systematic evaluation of the software in order to detect errors.
- **Walkthrough:** An informal review technique used to evaluate the product.
- **Formal Technical Review (FTR):** A formal review technique that is performed by a review team.

14.11 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short Answer Questions

1. Define software quality.
2. What are the different types of quality costs?
3. What are McCall's quality factors?

Long Answer Questions

1. What is software quality assurance? Explain.
2. Explain the various activities of SQA.
3. What do you understand by software reviews? Discuss its significance.
4. Explain the commonly used software quality approaches.
5. Explain the term software reliability.

NOTES

14.12 FURTHER READINGS

Pressman, Roger S. 1997. *Software Engineering, a Practitioner's Approach*. New Delhi: Tata McGraw-Hill.

Jalote, Pankaj. 1991. *An Integrated Approach to Software Engineering*. New Delhi: Narosa Publishing House.

Schach, Stephen R. 2005. *Object Oriented and Classical Software Engineering*. New Delhi: Tata McGraw-Hill.

Somerville, Ian. 2001. *Software Engineering*. New Delhi: Pearson Education.

Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandriolli . 1991. *Fundamentals of Software Engineering*. New Delhi: Prentice-Hill of India.

Jawadekar, Waman S. 2004. *Software Engineering: Principles and Practice*. New Delhi: Tata McGraw-Hill.